```
using System;

public class Program
{
    static void Main(string[] args)
    {
        int sum = Add(3, 5);
    }   Console.WriteLine(sum);

public static int Add(int a, int b)
{
    int result = a + b;
    return result;
}
```

INICIACIÓN C#

LUIS MINGUILLÓN PASCUAL

Prefacio

Este libro nace con la intención de acompañar a quienes desean dar sus primeros pasos en el mundo de la programación, utilizando C# como lenguaje de iniciación. C# es un lenguaje moderno, robusto y versátil, utilizado tanto en el desarrollo de aplicaciones de escritorio como en videojuegos, aplicaciones móviles y servicios web. Pero más allá de su potencia técnica, es también un lenguaje amigable para quienes comienzan a programar desde cero.

Mi propia andadura en el mundo de la programación ha sido un camino lleno de aprendizajes, errores, descubrimientos y satisfacciones. Empecé con lenguajes más sencillos, tanteando entre lógica, algoritmos y estructuras de control, sin saber muy bien a dónde me llevaría todo aquello. Con el tiempo, fui ampliando mis conocimientos y explorando diferentes tecnologías. Y cuando descubrí C#, sentí que había encontrado una herramienta poderosa, clara y estructurada, con la que podía construir cosas reales y útiles.

Este libro está pensado precisamente para quienes están donde yo estuve al comenzar: con ganas de aprender, pero sin saber por dónde empezar. Cada capítulo ha sido escrito con la intención de enseñar de forma progresiva, práctica y accesible, sin dar por sentado conocimientos previos. Si consigo que al terminar este libro puedas mirar tu primer programa con orgullo y sigas queriendo aprender más, habré cumplido mi objetivo.

Quiero hacer un agradecimiento muy especial a mi mujer, Marisa. Su apoyo constante, su paciencia en los momentos de duda, y su fe en mí incluso cuando yo no la tenía, han sido fundamentales para que este proyecto vea la luz. Este libro no existiría sin su presencia y su amor incondicional.

Gracias por acompañarme en este viaje.

Luis Minguillón Pascual

Índice del Curso de C# iniciación

Parte I – Fundamentos de C#

Lección 1: Fundamentos de C# - Introducción al Lenguaje C#

---

1. Introducción al Lenguaje C#

1.1 Historia y Evolución del Lenguaje C#

C# (pronunciado "C sharp") es un lenguaje de programación desarrollado por **Microsoft** a principios de los años 2000. Fue diseñado por un equipo liderado por **Anders Hejlsberg**, quien también participó en la creación de **Turbo Pascal** y **Delphi**.

Origen

- C# fue creado como parte del proyecto **.NET Framework**, con el objetivo de proporcionar un lenguaje moderno, orientado a objetos y fácil de usar para construir aplicaciones en Windows.
- Apareció por primera vez en **2000**, y su primera versión oficial fue **C# 1.0 en 2002**, junto con .NET Framework 1.0.
- Desde entonces ha evolucionado considerablemente, con versiones más modernas que incorporan características como programación asíncrona, expresiones lambda, patrones, y más.

Versiones importantes

Versión	Año	Características destacadas
C# 1.0	2002	Clases, interfaces, excepciones
C# 2.0	2005	Generics, métodos anónimos
C# 3.0	2007	LINQ, expresiones lambda
C# 5.0	2012	async/await
C# 6.0	2015	Interpolación de cadenas
C# 8.0	2019	Tipos de referencia anulables, patrones
C# 9.0	2020	Records, init-only properties
C# 10.0 y 11.0	2021–2022	Mejoras en patrones, clases parciales, interpolación avanzada

1.2 Características Principales de C#

C# es un lenguaje **moderno, seguro y potente**, que incluye lo mejor de varios paradigmas de programación.

Principales características

1. **Lenguaje orientado a objetos**: Todo en C# se basa en objetos (clases, herencia, polimorfismo).
2. **Sintaxis clara y estructurada**: Influenciado por C, C++ y Java.
3. **Seguro y tipado estáticamente**: Se detectan errores en tiempo de compilación.
4. **Multiplataforma**: Gracias a .NET Core y .NET 5/6+, C# se puede usar en Windows, Linux, macOS.
5. **Soporte para desarrollo moderno**:
   - LINQ (consultas a datos)
   - Programación asincrónica (async/await)
   - Expresiones lambda
6. **Interoperabilidad**: Se puede combinar con código en otros lenguajes como C++, VB.NET.
7. **Automatización de memoria**: Garbage Collector automático.

Ejemplo simple

```
using System;

class Programa
{
    static void Main()
    {
        Console.WriteLine("¡Hola, mundo!");
    }
}
```

1.3 Comparación con Otros Lenguajes

A continuación se comparan C# con otros lenguajes populares:

C# vs Java

Característica	C#	Java
Plataforma	.NET	JVM
Tipado	Estático	Estático
Propiedad	Microsoft	Oracle
LINQ	Sí	No
Interoperabilidad	Alta con COM/.NET	Limitada
Expresiones lambda	Sí	Sí (desde Java 8)
Programación funcional	Moderado (desde C# 3.0)	Limitado

C# vs Python

Característica	C#	Python
Tipado	Estático	Dinámico

Característica	C#	Python
Velocidad	Rápido (compilado)	Más lento (interpretado)
Sintaxis	Similar a C/Java	Más simple, legible
Curva de aprendizaje	Media	Baja
Paradigma	OO, funcional	Multiparadigma

C# vs C++

Característica	C#	C++
Gestión de memoria	Automática (GC)	Manual
Complejidad	Media	Alta
Plataforma	.NET, multiplataforma	Compilación nativa
Seguridad	Más seguro	Más bajo nivel

Actividad práctica

Ejercicio 1: Escribe un programa en C# que muestre tu nombre, tu edad y tu ciudad en tres líneas distintas.

```
using System;

class Programa
{
    static void Main()
    {
        Console.WriteLine("Nombre: Ana");
        Console.WriteLine("Edad: 30");
        Console.WriteLine("Ciudad: Zaragoza");
    }
}
```

Ejercicio 2: Investiga y escribe una breve biografía de Anders Hejlsberg.

Tarea sugerida

1. Crea una tabla comparativa entre C#, Python y Java con al menos 5 criterios (tipado, compilación, soporte OO, librerías, sintaxis).
2. Crea un programa en C# que muestre el año actual y un mensaje personalizado con `DateTime.Now.Year`.

```
using System;

class Programa
{
    static void Main()
    {
        Console.WriteLine("Estamos en el año " + DateTime.Now.Year);
        Console.WriteLine("¡Bienvenida al mundo de C#!");
    }
}
```

Autoevaluación

Contesta las siguientes preguntas:

1. ¿Quién diseñó el lenguaje C#?
2. ¿En qué año se lanzó C# oficialmente?
3. ¿Qué paradigmas admite C#?
4. ¿Qué ventaja ofrece LINQ?
5. ¿Cuál es la principal diferencia entre C# y Python en cuanto a tipado?

Conclusión

C# es un lenguaje moderno y robusto que combina la potencia de C++ con la simplicidad de lenguajes como Java o Python. Gracias a su evolución constante, es ideal para desarrollar aplicaciones de escritorio, web, móviles, videojuegos (Unity) y mucho más. Entender su historia y sus características te permitirá aprovechar todo su potencial desde el inicio.

Lección 2: Primeros pasos con C#

Objetivo:

Al finalizar esta lección, serás capaz de:

- Instalar y configurar Visual Studio correctamente.
- Comprender la estructura básica de un programa en C#.
- Compilar y ejecutar tu primer programa.

2.1 Instalación de Visual Studio

¿Qué es Visual Studio?

Visual Studio es un entorno de desarrollo integrado (IDE) creado por Microsoft. Ofrece soporte completo para el desarrollo en C#, incluyendo depuración, diseño de interfaces gráficas, integración con Git, herramientas de pruebas, y más.

Requisitos del sistema

- **Sistema operativo:** Windows 10 o superior
- **RAM:** Mínimo 4 GB (8 GB recomendado)
- **Espacio en disco:** Al menos 20 GB libres
- **Conexión a internet:** Necesaria para la instalación

Pasos para instalar Visual Studio:

1. **Descarga el instalador:**
    - Ve al sitio oficial: https://visualstudio.microsoft.com
    - Haz clic en “Descargar Visual Studio” → Elige **Visual Studio Community 2022 o 2025 (gratis)**
2. **Ejecuta el instalador:**
    - Se abrirá el **Visual Studio Installer**
3. **Selecciona las cargas de trabajo necesarias:**
    - Marca la opción **".NET desktop development"**
    - Opcionalmente, puedes agregar:
        - “Desarrollo multiplataforma con .NET MAUI”
        - “Desarrollo web con ASP.NET y .NET Core”
4. **Instala y espera:**

- Haz clic en “Instalar”.
- El proceso puede tardar entre 10 y 30 minutos dependiendo de tu conexión y equipo.

5. **Inicia Visual Studio:**
   - Al finalizar, puedes abrir Visual Studio.
   - Inicia sesión con una cuenta Microsoft (opcional, pero recomendado).
   - Selecciona el tema visual (oscuro, claro, etc.)

2.2 Estructura de un programa en C#

Estructura básica

```
using System;

namespace MiPrimerPrograma
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("¡Hola, mundo!");
        }
    }
}
```

Explicación paso a paso

Elemento	Explicación
`using System;`	Importa el espacio de nombres `System`, que contiene clases como `Console`.
`namespace MiPrimerPrograma`	Define un espacio de nombres para organizar el código.
`class Program`	Define una clase llamada `Program`, punto de entrada del programa.
`static void Main(string[] args)`	Método principal: es donde comienza la ejecución del programa.
`Console.WriteLine(...)`	Imprime texto en la consola.

Reglas básicas de sintaxis

- Toda instrucción finaliza con `;`
- Las llaves `{}` delimitan bloques de código.
- C# es **case sensitive**: `Main` ≠ `main`.

2.3 Compilación y ejecución

Crear un nuevo proyecto en Visual Studio

1. Abre Visual Studio.
2. Haz clic en **"Crear un nuevo proyecto"**.
3. Selecciona **Aplicación de Consola (.NET Core o .NET 6/7)**.
4. Asigna un nombre (por ejemplo, `HolaMundo`) y una ubicación.
5. Haz clic en **Crear**.

Escribir el código

Una vez creado el proyecto, verás un archivo llamado `Program.cs`. Reemplaza el contenido con el siguiente código:

```
using System;

namespace HolaMundo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("¡Hola, mundo desde C#!");
        }
    }
}
```

Ejecutar el programa

Hay dos formas principales:

- **Modo rápido (Debug):**
  Presiona `Ctrl + F5` o haz clic en el botón **Iniciar sin depurar**.
- **Modo con depuración:**
  Presiona `F5` o haz clic en **Iniciar depuración** para ver cómo se ejecuta paso a paso.

Resultado esperado

```
¡Hola, mundo desde C#!
Presione cualquier tecla para continuar...
```

Errores comunes y cómo solucionarlos

Error	Causa	Solución
`CS1002: ; expected`	Falta un punto y coma `;` al final de una línea	Añade el `;`
`CS1513: } expected`	Falta una llave de cierre	Verifica el balance de llaves `{}`
`Console.WriteLine no existe`	Falta el `using System;`	Asegúrate de tener `using System;` arriba

Actividad práctica

Ejercicio 1: Modifica el programa

Cambia el mensaje del programa para que imprima:

```
Hola, [Tu nombre], bienvenido a C#.
```

Ejemplo:

```
Console.WriteLine("Hola, Ana, bienvenida a C#.");
```

Ejercicio 2: Escribe múltiples líneas

Haz que el programa imprima 3 líneas con tus hobbies:

```
Console.WriteLine("Me gusta programar.");
Console.WriteLine("También me gusta leer.");
Console.WriteLine("Y disfruto tocar música.");
```

Resumen de la lección

- Instalamos Visual Studio y seleccionamos las cargas de trabajo adecuadas.
- Aprendimos la estructura básica de un programa en C#.
- Compilamos y ejecutamos nuestro primer programa desde Visual Studio.

Tarea opcional

1. Investiga qué hace el método `Console.ReadLine()`.
2. Crea un programa que pregunte tu nombre y lo salude:

```
Console.WriteLine("¿Cómo te llamas?");
string nombre = Console.ReadLine();
Console.WriteLine("Hola, " + nombre + ", ¡bienvenido!");
```

Lección 3: Sintaxis Básica en C#

Introducción

En esta lección aprenderás los elementos fundamentales de la sintaxis en C#. Estos son los bloques esenciales que necesitas para empezar a escribir programas funcionales. Esta base te permitirá comprender los temas más complejos en el futuro.

---

1. Tipos de datos primitivos

Los **tipos de datos primitivos** son los más básicos y permiten representar números, texto, valores lógicos, entre otros.

Tipo	Tamaño (bits)	Rango / Descripción	Ejemplo
`int`	32	Números enteros	`int edad = 25;`
`float`	32	Decimales con precisión simple	`float pi = 3.14f;`
`double`	64	Decimales con doble precisión	`double d = 3.14159;`
`char`	16	Un solo carácter Unicode	`char letra = 'A';`
`string`	Depende	Cadena de caracteres	`string nombre = "Ana";`
`bool`	1 (interno)	Lógico: `true` o `false`	`bool activo = true;`
`decimal`	128	Alta precisión decimal (finanzas)	`decimal saldo = 1999.99m;`
`byte`	8	Entero sin signo (0 a 255)	`byte b = 200;`

Nota: `float` y `decimal` requieren sufijos (`f`, `m`).

---

2. Variables y constantes

Declaración de variables

Una **variable** es un espacio en memoria que almacena un valor que puede cambiar durante la ejecución.

```
int edad = 30;

float altura = 1.75f;

string nombre = "Lucía";
```

Reglas para nombrar variables

- Deben comenzar con letra o guion bajo (_), no con número.
- No pueden contener espacios ni caracteres especiales (excepto _).
- Son sensibles a mayúsculas y minúsculas: `edad` ≠ `Edad`.
- No se puede usar una palabra clave de C# como nombre.

Constantes

Una **constante** es un valor que no puede modificarse después de su declaración.

```
const double PI = 3.14159;

const string MENSAJE = "Hola mundo";
```

Nota: Las constantes deben inicializarse en la misma línea en la que se declaran.

---

3. Comentarios

Los **comentarios** sirven para documentar el código. No afectan la ejecución.

Comentario de una línea

```
// Esto es un comentario de una línea

int edad = 25; // Esta variable almacena la edad
```

Comentario de múltiples líneas

```
/*

 Este es un comentario

 de varias líneas

*/
```

---

4. Operadores

Operadores aritméticos

Se usan para realizar operaciones matemáticas.

Operador	Descripción	Ejemplo
`+`	Suma	`a + b`
`-`	Resta	`a - b`
`*`	Multiplicación	`a * b`
`/`	División	`a / b`
`%`	Módulo (residuo)	`a % b`

```
int a = 10, b = 3;

Console.WriteLine(a + b); // 13

Console.WriteLine(a % b); // 1
```

Operadores de comparación

Devuelven un valor `bool` (`true` o `false`).

Operador	Significado	Ejemplo
`==`	Igual a	`a == b`

Operador	Significado	Ejemplo
!=	Distinto de	a != b
<	Menor que	a < b
>	Mayor que	a > b
<=	Menor o igual que	a <= b
>=	Mayor o igual que	a >= b

```
int x = 5, y = 10;

Console.WriteLine(x < y); // True

Console.WriteLine(x == y); // False
```

Operadores lógicos

Trabajan con valores booleanos.

Operador	Nombre	Ejemplo	Resultado
&&	AND lógico	true && false	false
`		`	OR lógico
!	NOT lógico	!true	false

```
bool a = true;

bool b = false;


Console.WriteLine(a && b); // False

Console.WriteLine(a || b); // True

Console.WriteLine(!a);     // False
```

Ejemplo completo

```
using System;


class Program

{

    static void Main()

    {

        // Declaración de variables

        int edad = 28;

        float altura = 1.68f;

        bool esMayor = edad >= 18;

        string nombre = "María";


        // Comentarios y operadores

        Console.WriteLine("Nombre: " + nombre);
```

```
        Console.WriteLine("Edad: " + edad);
        Console.WriteLine("¿Es mayor de edad? " + esMayor);
        Console.WriteLine("Altura al cuadrado: " + (altura * altura));

        // Operadores lógicos
        bool tieneDni = true;
        bool puedeEntrar = esMayor && tieneDni;
        Console.WriteLine("¿Puede entrar? " + puedeEntrar);
    }
}
```

Ejercicios propuestos

1. Declara una variable para tu edad, tu estatura y si tienes carnet de conducir. Imprime un mensaje como:

   ```
   Tengo 25 años, mido 1.75 y ¿tengo carnet? True
   ```

2. Crea un programa que determine si dos números son iguales o distintos.
3. Escribe un programa que verifique si un número es par o impar usando el operador `%`.
4. Dado el valor de dos booleanos `a` y `b`, imprime el resultado de todas las combinaciones posibles con los operadores lógicos &&, `||` y `!`.

Conclusión

Dominar la sintaxis básica en C# es esencial para seguir aprendiendo. Comprender los tipos primitivos, cómo declarar variables y constantes, usar operadores y documentar con comentarios sienta las bases para programar estructuras más complejas como condicionales, bucles y clases.

Lección 4: Estructuras de Control en C#

Objetivos de la lección

- Comprender y utilizar las estructuras condicionales `if`, `else` y `switch`.
- Dominar los bucles `for`, `while` y `do while`.
- Aprender a controlar el flujo de ejecución con `break`, `continue` y `goto`.

---

1. Introducción

Las **estructuras de control** permiten modificar el flujo de ejecución de un programa, lo que nos permite tomar decisiones y repetir bloques de código.

---

2. Sentencias Condicionales

2.1 `if` y `else`

Sirven para ejecutar un bloque de código si se cumple una condición.

```
int edad = 18;

if (edad >= 18)
{
    Console.WriteLine("Eres mayor de edad.");
}
else
{
    Console.WriteLine("Eres menor de edad.");
}
```

Variaciones:

- `if` solo.
- `if` + `else`.
- `if` + `else if` + `else`.

```
int nota = 85;

if (nota >= 90)
{
    Console.WriteLine("Sobresaliente");
```

```
}
else if (nota >= 70)
{
    Console.WriteLine("Aprobado");
}
else
{
    Console.WriteLine("Reprobado");
}
```

2.2 `switch`

Sirve para múltiples comparaciones sobre una misma variable.

```
int dia = 3;

switch (dia)
{
    case 1:
        Console.WriteLine("Lunes");
        break;
    case 2:
        Console.WriteLine("Martes");
        break;
    case 3:
        Console.WriteLine("Miércoles");
        break;
    default:
        Console.WriteLine("Día no válido");
        break;
}
```

Notas:

- Cada `case` debe terminar con `break` para evitar que continúe al siguiente.
- `default` es el caso por defecto.

---

3. Bucles

Los bucles permiten repetir código mientras se cumpla una condición.

3.1 Bucle `for`

Útil cuando se conoce cuántas veces se va a repetir.

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine("Iteración: " + i);
}
```

3.2 Bucle `while`

Repite mientras la condición sea verdadera.

```
int contador = 0;

while (contador < 5)
{
    Console.WriteLine("Contador: " + contador);
    contador++;
}
```

3.3 Bucle `do while`

Ejecuta primero y luego evalúa la condición.

```
int numero = 1;

do
{
    Console.WriteLine("Número: " + numero);
    numero++;
}
while (numero <= 5);
```

---

4. Control del Flujo de Ejecución

4.1 `break`

Sale del bucle o `switch` inmediatamente.

```
for (int i = 0; i < 10; i++)
{
    if (i == 5)
```

```
        break;

    Console.WriteLine("i = " + i);
}
```

4.2 `continue`

Salta el resto del bucle y continúa con la siguiente iteración.

```
for (int i = 0; i < 10; i++)
{
    if (i % 2 == 0)
        continue;

    Console.WriteLine("Impar: " + i);
}
```

4.3 `goto`

Salta a una etiqueta específica en el código (desaconsejado en la mayoría de los casos).

```
int numero = 1;

inicio:
Console.WriteLine("Número: " + numero);
numero++;

if (numero <= 5)
    goto inicio;
```

---

5. Ejercicios Propuestos

Ejercicio 1: Clasificador de edades

Pide al usuario su edad y muestra si es niño (<13), adolescente (13-17), adulto (18-59), o anciano (60+).

Ejercicio 2: Tabla de multiplicar

Solicita un número y muestra su tabla de multiplicar del 1 al 10 usando un bucle `for`.

Ejercicio 3: Contador de pares e impares

Muestra cuántos números pares e impares hay del 1 al 50 usando `while`.

Ejercicio 4: Menú con `switch`

Crea un menú simple:

```
1. Saludar
2. Mostrar fecha
3. Salir
```

Usa `switch` para ejecutar cada acción.

Ejercicio 5: Bucle infinito con `break`

Crea un bucle que pide nombres al usuario hasta que escriba "salir".

---

6. Buenas prácticas

- Prefiere `if` claros antes que anidados o muy complejos.
- Usa `switch` para múltiples comparaciones sobre una misma variable.
- Evita `goto` salvo que tengas una justificación muy clara.
- Usa `break` y `continue` con moderación y claridad.

---

7. Conclusión

Las estructuras de control son la base para que tus programas puedan **tomar decisiones** y **repetir tareas automáticamente**. Combinándolas adecuadamente puedes resolver problemas complejos de forma ordenada y legible.

Lección 5: Completa de Programación en C#: Métodos y Funciones

Objetivo de la lección

Al finalizar esta lección, el estudiante será capaz de:

- Definir y utilizar métodos en C#.
- Utilizar parámetros y valores de retorno.
- Aplicar sobrecarga de métodos.
- Diferenciar y utilizar métodos estáticos.

---

1. ¿Qué es un método?

Un **método** (también llamado función) es un bloque de código que realiza una tarea específica y puede ser reutilizado. Sirve para **organizar, reutilizar y estructurar** el código de forma lógica.

En C#, todos los métodos están definidos dentro de una clase.

---

2. Declaración y uso de métodos

Sintaxis general

```
[modificador] [tipo_de_retorno] NombreDelMetodo([parámetros])
{
    // Código del método
    return valor; // opcional si tipo_de_retorno es void
}
```

Ejemplo básico

```
using System;

class Programa
{
    static void Saludar()
    {
        Console.WriteLine("Hola, bienvenido al curso de C#.");
    }

    static void Main()
```

```
    {
        Saludar(); // Llamada al método
    }
}
```

Explicación:

- `static` significa que pertenece a la clase, no a una instancia.
- `void` indica que no devuelve nada.
- `Saludar()` es el nombre del método.

---

3. Parámetros y retorno

Los **parámetros** permiten pasar información al método. El **valor de retorno** permite devolver un resultado al llamador.

Ejemplo con parámetros y retorno

```
using System;

class Calculadora
{
    static int Sumar(int a, int b)
    {
        return a + b;
    }

    static void Main()
    {
        int resultado = Sumar(3, 5);
        Console.WriteLine("Resultado: " + resultado);
    }
}
```

Explicación:

- `int a, int b` son los parámetros.
- El método devuelve un `int`.
- `return a + b` devuelve el resultado.

Parámetros por valor y por referencia

```
static void Incrementar(ref int numero)
{
    numero++;
}

static void Main()
{
    int x = 10;
    Incrementar(ref x);
    Console.WriteLine(x); // 11
}
```

- `ref` permite modificar el valor original.

---

4. Sobrecarga de métodos

La **sobrecarga de métodos** permite definir varios métodos con el **mismo nombre** pero **diferente número o tipo de parámetros**.

Ejemplo:

```
class Calculadora
{
    static int Sumar(int a, int b) => a + b;
    static double Sumar(double a, double b) => a + b;
    static int Sumar(int a, int b, int c) => a + b + c;

    static void Main()
    {
        Console.WriteLine(Sumar(2, 3));       // 5
        Console.WriteLine(Sumar(2.5, 3.5));   // 6.0
        Console.WriteLine(Sumar(1, 2, 3));    // 6
    }
}
```

Regla de oro:

El compilador distingue métodos sobrecargados por:

- Número de parámetros
- Tipo de parámetros

- Orden de parámetros

¡No se puede sobrecargar solo por tipo de retorno!

---

5. Métodos estáticos

Un **método estático** pertenece a la **clase** en lugar de a una **instancia del objeto**.

Uso común:

- Métodos auxiliares
- Utilidades (por ejemplo, `Math.Sqrt()`)

Ejemplo:

```
class Utilidad
{
    public static void MostrarFecha()
    {
        Console.WriteLine(DateTime.Now);
    }
}

class Programa
{
    static void Main()
    {
        Utilidad.MostrarFecha();
    }
}
```

Ventaja:

No necesitas crear un objeto para usar un método estático.

Ejercicios prácticos

Ejercicio 1

Crear un método llamado `EsPar` que reciba un número entero y devuelva `true` si es par, `false` si no.

Solución:

```
static bool EsPar(int numero)
{
```

```
    return numero % 2 == 0;
}
```

Ejercicio 2

Sobrecargar un método llamado `MostrarMensaje` que imprima:

- Solo un mensaje
- Un mensaje y una cantidad de veces a repetirlo

Solución:

```
static void MostrarMensaje(string mensaje)
{
    Console.WriteLine(mensaje);
}

static void MostrarMensaje(string mensaje, int veces)
{
    for (int i = 0; i < veces; i++)
        Console.WriteLine(mensaje);
}
```

Ejercicio 3

Crear un método `Factorial(int n)` que devuelva el factorial de un número.

Solución:

```
static long Factorial(int n)
{
    long resultado = 1;
    for (int i = 2; i <= n; i++)
        resultado *= i;
    return resultado;
}
```

Buenas prácticas

- Usa nombres descriptivos para métodos.
- Métodos deben realizar **una única tarea clara**.
- Evita métodos demasiado largos.

- Si un método es común, considera hacerlo `static`.
- Documenta con comentarios si el comportamiento no es obvio.

Repaso Final

Concepto	Descripción breve
Método	Función definida dentro de una clase
Parámetro	Entrada al método
Retorno	Valor devuelto por el método
Sobrecarga	Mismo nombre, distintos parámetros
Método estático	Pertenece a la clase, no requiere instancia

Lección 6: Arrays y Colecciones en C#

Objetivos de la lección

- Comprender el uso de arrays unidimensionales y multidimensionales.
- Utilizar colecciones genéricas como List<T>, Dictionary<TKey, TValue>, `Queue`, y `Stack`.
- Recorrer colecciones con `foreach`.
- Aplicar operaciones comunes con colecciones.

---

1. Arrays en C#

1.1 Arrays Unidimensionales

Los **arrays** (o arreglos) son estructuras de datos que almacenan elementos del mismo tipo en posiciones contiguas de memoria.

```
int[] numeros = new int[5]; // array con 5 elementos
numeros[0] = 10;
numeros[1] = 20;
// ...
```

También pueden inicializarse directamente:

```
string[] nombres = { "Ana", "Luis", "Pedro" };
Console.WriteLine(nombres[1]); // Luis
```

Los índices comienzan desde 0.

1.2 Arrays Multidimensionales

2D (matrices)

```
int[,] matriz = new int[2, 3]
{
    {1, 2, 3},
    {4, 5, 6}
};

Console.WriteLine(matriz[1, 2]); // 6
```

3D o más dimensiones

```
int[,,] cubo = new int[2, 2, 2];
cubo[0, 0, 0] = 1;
```

---

2. Colecciones Genéricas

Las colecciones permiten almacenar objetos de forma dinámica.

2.1 `List<T>`

Lista dinámica que puede crecer o disminuir.

```
List<string> frutas = new List<string>();
frutas.Add("Manzana");
frutas.Add("Pera");

Console.WriteLine(frutas[0]); // Manzana
```

Métodos comunes:

- `Add()`, `Remove()`, `Insert()`, `Contains()`, `Clear()`, `Count`

2.2 `Dictionary<TKey, TValue>`

Almacena pares clave-valor.

```
Dictionary<string, int> edades = new Dictionary<string, int>();
edades["Ana"] = 30;
edades["Luis"] = 25;

Console.WriteLine(edades["Ana"]); // 30
```

Métodos comunes:

- `Add()`, `Remove()`, `ContainsKey()`, `TryGetValue()`

2.3 `Queue<T>`

Cola: estructura FIFO (First In, First Out).

```
Queue<string> cola = new Queue<string>();
cola.Enqueue("Primero");
cola.Enqueue("Segundo");

Console.WriteLine(cola.Dequeue()); // Primero
```

Métodos:

- `Enqueue()`, `Dequeue()`, `Peek()`, `Clear()`, `Count`

2.4 `Stack<T>`

Pila: estructura LIFO (Last In, First Out).

```
Stack<string> pila = new Stack<string>();
pila.Push("Uno");
pila.Push("Dos");

Console.WriteLine(pila.Pop()); // Dos
```

Métodos:

- `Push()`, `Pop()`, `Peek()`, `Count`, `Clear()`

---

3. Iteración con `foreach`

Ideal para recorrer colecciones.

```
string[] colores = { "Rojo", "Verde", "Azul" };
foreach (string color in colores)
{
    Console.WriteLine(color);
}
```

También funciona con Listas, Diccionarios, Pilas y Colas:

```
List<int> numeros = new List<int> { 1, 2, 3 };
foreach (int n in numeros)
{
    Console.WriteLine(n);
}
```

---

4. Operaciones comunes con colecciones

4.1 Buscar elementos

```
if (frutas.Contains("Pera"))
    Console.WriteLine("La lista contiene Pera.");
```

4.2 Ordenar listas

```
frutas.Sort(); // orden alfabético
```

4.3 Eliminar elementos

```
frutas.Remove("Pera");
```

4.4 Convertir arrays en listas

```
int[] arreglo = { 1, 2, 3 };
List<int> lista = new List<int>(arreglo);
```

Ejemplo completo

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        List<string> tareas = new List<string> { "Estudiar", "Cocinar",
"Leer" };

        Console.WriteLine("Tareas:");
        foreach (string tarea in tareas)
        {
            Console.WriteLine("- " + tarea);
        }

        Dictionary<string, int> inventario = new Dictionary<string, int>();
        inventario["Lápices"] = 10;
        inventario["Cuadernos"] = 5;

        Console.WriteLine("\nInventario:");
        foreach (var item in inventario)
        {
            Console.WriteLine($"{item.Key}: {item.Value}");
        }

        Queue<string> fila = new Queue<string>();
        fila.Enqueue("Cliente 1");
        fila.Enqueue("Cliente 2");

        Console.WriteLine($"\nAtendiendo: {fila.Dequeue()}");
```

```
        Stack<string> historial = new Stack<string>();
        historial.Push("Página A");
        historial.Push("Página B");

        Console.WriteLine($"\nÚltima página visitada: {historial.Pop()}");
    }
}
```

Ejercicios prácticos

1. **Array de temperaturas**: Crea un array de 7 enteros y muestra la media.
2. **Diccionario de estudiantes**: Almacena nombres y notas. Imprime los que aprobaron.
3. **Simulador de cola de banco**: Usa `Queue<string>` para simular atención a clientes.
4. **Historial de acciones**: Usa `Stack<string>` para guardar acciones del usuario y deshacer la última.
5. **Lista de tareas**: Permite agregar, eliminar y mostrar tareas con `List<string>`.

Resumen final

Estructura	Tipo	Uso principal
`Array`	Fijo	Almacenar elementos iguales
`List<T>`	Dinámico	Listas modificables
`Dictionary<>`	Asociación	Clave-valor
`Queue<T>`	FIFO	Procesos en orden
`Stack<T>`	LIFO	Deshacer acciones

Parte II – Programación Orientada a Objetos

Lección 7: Clases y Objetos en C#

Objetivos de la lección

- Comprender qué son las clases y los objetos en C#.
- Aprender a definir clases y crear objetos.
- Entender los conceptos de campos, propiedades, métodos y constructores.
- Aplicar encapsulamiento.
- Utilizar clases para modelar entidades del mundo real.
- Realizar ejercicios para afianzar lo aprendido.

---

1. ¿Qué es una clase?

Una **clase** es una plantilla o molde que define las características y comportamientos de un tipo de objeto. En otras palabras, **una clase define qué puede tener y qué puede hacer un objeto**.

Sintaxis básica:

```
class NombreClase
{
    // Campos (variables internas)
    // Propiedades
    // Métodos
    // Constructores
}
```

---

2. ¿Qué es un objeto?

Un **objeto** es una instancia concreta de una clase. Cuando creas un objeto, estás creando una **copia real y funcional** de la clase.

Ejemplo:

```
// Clase
class Persona
{
    public string nombre;
```

```
    public int edad;
}

// Crear objeto
Persona p1 = new Persona();
p1.nombre = "Ana";
p1.edad = 30;
```

---

3. Campos (fields)

Son variables declaradas dentro de la clase que representan el estado de un objeto.

```
class Persona
{
    public string nombre;
    public int edad;
}
```

---

4. Propiedades (properties)

Permiten acceder y modificar campos de forma segura. Usan métodos internos `get` y `set`.

```
class Persona
{
    private int edad;

    public int Edad
    {
        get { return edad; }
        set
        {
            if (value >= 0) edad = value;
        }
    }
}
```

---

5. Métodos

Son funciones dentro de una clase que realizan acciones.

```
class Persona
```

```
{
    public string nombre;

    public void Saludar()
    {
        Console.WriteLine("Hola, soy " + nombre);
    }
}
```

---

6. Constructores

Son métodos especiales que se ejecutan cuando se crea un objeto. Se llaman igual que la clase y no tienen tipo de retorno.

```
class Persona
{
    public string nombre;

    public Persona(string nombre)
    {
        this.nombre = nombre;
    }
}
```

---

7. Encapsulamiento

El **encapsulamiento** permite ocultar detalles internos del objeto y exponer solo lo necesario. Se logra usando `private`, `public`, `protected`.

```
class CuentaBancaria
{
    private double saldo;

    public double Saldo
    {
        get { return saldo; }
        private set { saldo = value; }
    }

    public void Depositar(double cantidad)
```

```
    {
        if (cantidad > 0) saldo += cantidad;
    }
}
```

---

8. Ejemplo completo

```
using System;

class Coche
{
    private string marca;
    private int año;

    public Coche(string marca, int año)
    {
        this.marca = marca;
        this.año = año;
    }

    public void MostrarInfo()
    {
        Console.WriteLine($"Marca: {marca}, Año: {año}");
    }

    public void Encender()
    {
        Console.WriteLine("El coche está encendido.");
    }
}

class Programa
{
    static void Main()
    {
        Coche miCoche = new Coche("Toyota", 2020);
        miCoche.MostrarInfo();
```

```
        miCoche.Encender();
    }
}
```

---

9. Buenas prácticas

- Nombres de clases en PascalCase.
- Los campos deben ser privados; usa propiedades para acceder a ellos.
- Inicializa tus objetos correctamente con constructores.
- Crea métodos que representen bien el comportamiento del objeto.

---

10. Ejercicios propuestos

Ejercicio 1: Clase Libro

Crea una clase `Libro` con campos `titulo`, `autor`, `anio`, y un método `MostrarInfo`.

Ejercicio 2: Clase Rectángulo

Crea una clase `Rectangulo` con base y altura. Añade métodos para calcular el área y perímetro.

Ejercicio 3: Clase CuentaBancaria

Crea una clase con `Depositar`, `Retirar` y `MostrarSaldo`, usando encapsulamiento.

---

11. Soluciones

Solución Ejercicio 1

```
class Libro
{
    public string titulo;
    public string autor;
    public int anio;

    public void MostrarInfo()
    {
        Console.WriteLine($"{titulo}, de {autor}, publicado en {anio}.");
    }
}
```

Solución Ejercicio 2

```
class Rectangulo
{
    public double baseRect;
    public double altura;

    public double CalcularArea()
    {
        return baseRect * altura;
    }

    public double CalcularPerimetro()
    {
        return 2 * (baseRect + altura);
    }
}
```

Solución Ejercicio 3

```
class CuentaBancaria
{
    private double saldo;

    public void Depositar(double cantidad)
    {
        if (cantidad > 0) saldo += cantidad;
    }

    public void Retirar(double cantidad)
    {
        if (cantidad > 0 && cantidad <= saldo) saldo -= cantidad;
    }

    public void MostrarSaldo()
    {
        Console.WriteLine($"Saldo: {saldo}");
    }
}
```

---

12. Conclusión

Las clases y los objetos son el núcleo de la programación orientada a objetos. Dominar estos conceptos te permitirá construir programas más organizados, reutilizables y fáciles de mantener.

¿Te gustaría que prepare un PDF con esta lección o que lo convierta en una serie de ejercicios guiados paso a paso?

Lección 8: Herencia en C#

Objetivo de la lección

Al finalizar esta lección, serás capaz de:

- Comprender qué es la herencia y cómo se aplica en C#.
- Crear clases base y derivadas correctamente.
- Usar el modificador `base` para invocar constructores y métodos de la clase padre.
- Aplicar la sobreescritura de métodos (`override` y `virtual`).
- Comprender el uso de `sealed`, `abstract` y `new` en el contexto de herencia.

---

1. ¿Qué es la Herencia?

La **herencia** es un principio fundamental de la programación orientada a objetos que permite crear una nueva clase (**derivada**) a partir de otra clase existente (**base**). La clase derivada hereda todos los campos, propiedades y métodos públicos y protegidos de la clase base, pudiendo además extenderlos o modificarlos.

Sintaxis básica:

```
class ClaseBase {
    public void MetodoBase() {
        Console.WriteLine("Método de la clase base");
    }
}

class ClaseDerivada : ClaseBase {
    public void MetodoDerivado() {
        Console.WriteLine("Método de la clase derivada");
    }
}
```

---

2. Ejemplo práctico simple

```
using System;

class Animal {
    public string Nombre { get; set; }
```

```
    public void Respirar() {
        Console.WriteLine($"{Nombre} está respirando.");
    }
}

class Perro : Animal {
    public void Ladrar() {
        Console.WriteLine($"{Nombre} está ladrando.");
    }
}

class Programa {
    static void Main() {
        Perro miPerro = new Perro();
        miPerro.Nombre = "Fido";
        miPerro.Respirar(); // método heredado
        miPerro.Ladrar();   // método propio
    }
}
```

---

3. Uso del constructor base con `base()`

```
class Persona {
    public string Nombre;

    public Persona(string nombre) {
        Nombre = nombre;
    }
}

class Empleado : Persona {
    public int Id;

    public Empleado(string nombre, int id) : base(nombre) {
        Id = id;
    }
```

```
    public void Mostrar() {
        Console.WriteLine($"Nombre: {Nombre}, ID: {Id}");
    }
}
```

---

4. Sobreescritura de métodos: `virtual`, `override`, `new`

- **`virtual`: indica que un método puede ser sobreescrito.**
- **`override`: sobreescribe el comportamiento del método virtual.**
- **`new`: oculta el método de la clase base sin sobreescribirlo.**

```
class Vehiculo {
    public virtual void Encender() {
        Console.WriteLine("Vehículo encendido");
    }
}

class Coche : Vehiculo {
    public override void Encender() {
        Console.WriteLine("Coche encendido");
    }
}

class Moto : Vehiculo {
    public new void Encender() {
        Console.WriteLine("Moto encendida (ocultando método base)");
    }
}
```

---

5. Clases `abstract` y métodos `abstract`

Una clase `abstract` **no puede ser instanciada**, y puede contener métodos que **deben ser implementados** por las clases derivadas.

```
abstract class Figura {
    public abstract double CalcularArea();
}

class Cuadrado : Figura {
```

```
    public double Lado;

    public Cuadrado(double lado) {
        Lado = lado;
    }

    public override double CalcularArea() {
        return Lado * Lado;
    }
}
```

---

6. Modificador `sealed`

Evita que una clase pueda ser heredada.

```
sealed class Final {
    public void Mensaje() {
        Console.WriteLine("No se puede heredar esta clase.");
    }
}

// Esto causará un error:
// class SubFinal : Final { }
```

---

7. Ejercicios prácticos

Ejercicio 1: Clase `Empleado` y `Gerente`

Crea una clase `Empleado` con propiedades `Nombre`, `Sueldo`. Luego una clase `Gerente` que herede de `Empleado` y añada `Departamento`.

Solución:

```
class Empleado {
    public string Nombre;
    public double Sueldo;

    public void Mostrar() {
        Console.WriteLine($"Empleado: {Nombre}, Sueldo: {Sueldo}");
    }
}
```

```
class Gerente : Empleado {
    public string Departamento;

    public void MostrarDepartamento() {
        Console.WriteLine($"Departamento: {Departamento}");
    }
}

class Programa {
    static void Main() {
        Gerente g = new Gerente();
        g.Nombre = "Lucía";
        g.Sueldo = 4000;
        g.Departamento = "Ventas";

        g.Mostrar();
        g.MostrarDepartamento();
    }
}
```

Ejercicio 2: Método virtual `Hablar()`

Haz una clase `Animal` con un método virtual `Hablar()`. Luego crea `Gato` y `Perro` que sobreescriban este método.

Solución:

```
class Animal {
    public virtual void Hablar() {
        Console.WriteLine("Sonido genérico");
    }
}

class Gato : Animal {
    public override void Hablar() {
        Console.WriteLine("Miau");
    }
}
```

```
class Perro : Animal {
    public override void Hablar() {
        Console.WriteLine("Guau");
    }
}
```

---

8. Buenas prácticas

- Usa `virtual` y `override` solo si necesitas polimorfismo.
- Si no vas a heredar una clase, marca como `sealed`.
- Prefiere la composición a la herencia cuando no hay una clara relación "es-un".
- Usa nombres significativos que reflejen jerarquía y comportamiento.

Conclusiones

- La herencia promueve la reutilización del código.
- C# permite herencia simple: una clase puede heredar de una sola clase.
- Polimorfismo y herencia suelen ir de la mano.
- Usa `abstract` y `virtual` para definir comportamientos extensibles.

Lección 9: Polimorfismo en C#

Objetivos de la lección

- Comprender qué es el polimorfismo y por qué es importante en la POO.
- Diferenciar entre **polimorfismo en tiempo de compilación** y **polimorfismo en tiempo de ejecución**.
- Implementar ejemplos prácticos usando **métodos virtuales**, **override**, **abstract** y **interfaces**.
- Aplicar el concepto de polimorfismo para escribir código flexible, reutilizable y extensible.

¿Qué es el polimorfismo?

Polimorfismo significa "muchas formas". En el contexto de la Programación Orientada a Objetos (POO), permite que objetos de diferentes clases puedan ser tratados como objetos de una clase base común.

Esto permite:

- Ejecutar el **mismo método** en diferentes tipos de objetos.
- Usar **herencia** y **sobrescritura** de métodos para modificar comportamientos.

Tipos de Polimorfismo en C#

1. Polimorfismo en tiempo de compilación (Estático)

- También llamado **sobrecarga de métodos**.
- Permite definir varios métodos con el mismo nombre pero diferentes parámetros.

🔧 **Ejemplo:**

```
public class Calculadora
{
    public int Sumar(int a, int b)
    {
        return a + b;
    }

    public double Sumar(double a, double b)
    {
        return a + b;
    }
```

```
    public int Sumar(int a, int b, int c)
    {
        return a + b + c;
    }
}
```

---

2. Polimorfismo en tiempo de ejecución (Dinámico)

- Se logra con **herencia** y **métodos virtuales**/`override`.
- Permite que una llamada a un método se resuelva **en tiempo de ejecución**, dependiendo del tipo del objeto.

Claves:

- `virtual`: en la clase base, indica que el método puede ser sobrescrito.
- `override`: en la clase derivada, indica que se está sobrescribiendo el método.
- `abstract`: método sin implementación, obliga a sobrescribirlo en clases derivadas.
- `interface`: define métodos que las clases deben implementar, habilita polimorfismo por contrato.

Ejemplo con `virtual` y `override`

```
public class Animal
{
    public virtual void HacerSonido()
    {
        Console.WriteLine("El animal hace un sonido");
    }
}

public class Perro : Animal
{
    public override void HacerSonido()
    {
        Console.WriteLine("El perro ladra");
    }
}

public class Gato : Animal
```

```
{
    public override void HacerSonido()
    {
        Console.WriteLine("El gato maúlla");
    }
}
```

Uso:

```
Animal miAnimal = new Perro();
miAnimal.HacerSonido();  // Salida: "El perro ladra"

miAnimal = new Gato();
miAnimal.HacerSonido();  // Salida: "El gato maúlla"
```

Ejemplo con Clases Abstractas

```
public abstract class Figura
{
    public abstract double CalcularArea();
}

public class Circulo : Figura
{
    public double Radio { get; set; }

    public Circulo(double radio)
    {
        Radio = radio;
    }

    public override double CalcularArea()
    {
        return Math.PI * Radio * Radio;
    }
}

public class Rectangulo : Figura
{
```

```
    public double Ancho { get; set; }
    public double Alto { get; set; }

    public Rectangulo(double ancho, double alto)
    {
        Ancho = ancho;
        Alto = alto;
    }

    public override double CalcularArea()
    {
        return Ancho * Alto;
    }
}
```

Uso:

```
Figura figura = new Circulo(5);
Console.WriteLine(figura.CalcularArea());  // Área del círculo

figura = new Rectangulo(4, 6);
Console.WriteLine(figura.CalcularArea());  // Área del rectángulo
```

Polimorfismo con Interfaces

```
public interface IReproducible
{
    void Reproducir();
}

public class Cancion : IReproducible
{
    public void Reproducir()
    {
        Console.WriteLine("Reproduciendo canción...");
    }
}

public class Video : IReproducible
```

```
{
    public void Reproducir()
    {
        Console.WriteLine("Reproduciendo video...");
    }
}
```

Uso:

```
List<IReproducible> lista = new List<IReproducible>
{
    new Cancion(),
    new Video()
};

foreach (var item in lista)
{
    item.Reproducir();
}
```

Beneficios del Polimorfismo

- **Extensibilidad**: puedes agregar nuevos tipos sin cambiar el código existente.
- **Reutilización**: reduce la duplicación de código.
- **Mantenimiento**: facilita modificaciones sin romper el programa.
- **Abstracción**: los clientes trabajan con interfaces o clases base.

Ejercicio Propuesto

Crea una jerarquía de clases para representar empleados. La clase base debe ser `Empleado` con un método `CalcularSueldo()`. Heredan `EmpleadoFijo` y `EmpleadoPorHora`, cada uno con su propia implementación del sueldo.

Requisitos:

- Clase `Empleado`: método virtual `CalcularSueldo()`.
- `EmpleadoFijo`: tiene un sueldo fijo.
- `EmpleadoPorHora`: tiene tarifa por hora y horas trabajadas.

```
// Código de ejemplo para empezar
public abstract class Empleado
```

```
{
    public string Nombre { get; set; }
    public Empleado(string nombre)
    {
        Nombre = nombre;
    }

    public abstract double CalcularSueldo();
}
```

Puedes implementar las clases derivadas y probar con una lista de `Empleado` mostrando los sueldos.

Resumen Final

Concepto	Descripción
`virtual`	Declara un método que puede ser sobreescrito.
`override`	Reescribe un método virtual.
`abstract`	Declara un método sin implementación que debe sobrescribirse.
`interface`	Define un contrato que las clases deben implementar.
Polimorfismo dinámico	Usa clases base e invoca métodos que se resuelven en tiempo real.
Polimorfismo estático	Usa sobrecarga de métodos en la misma clase.

Lección 10: Constructores y Destructores en C#

Objetivos de la lección

- Comprender qué es un constructor y un destructor en C#
- Conocer los distintos tipos de constructores
- Aprender cómo se usan los destructores
- Aplicar constructores y destructores en clases reales
- Practicar con ejemplos y ejercicios

---

1. ¿Qué es un Constructor?

Un **constructor** es un **método especial** que se ejecuta automáticamente **al crear una instancia (objeto)** de una clase.

🔧 Sintaxis básica:

```
public class Persona
{
    public string Nombre;

    // Constructor
    public Persona()
    {
        Nombre = "Sin nombre";
    }
}
```

Características de un constructor

- Tiene el **mismo nombre** que la clase.
- No tiene tipo de retorno (ni `void`).
- Se ejecuta **una sola vez**, al crear el objeto.
- Puede tener **parámetros** (constructores sobrecargados).
- Se puede definir más de un constructor.

---

2. Tipos de Constructores

a. Constructor por defecto

Creador sin parámetros. Si no defines uno, el compilador lo crea por ti.

```
public class Vehiculo
{
    public string Marca;

    public Vehiculo()
    {
        Marca = "Desconocida";
    }
}
```

---

b. Constructor parametrizado

Permite inicializar objetos con valores específicos.

```
public class Vehiculo
{
    public string Marca;

    public Vehiculo(string marca)
    {
        Marca = marca;
    }
}
```

---

c. Sobrecarga de constructores

Puedes tener varios constructores con diferentes parámetros:

```
public class Libro
{
    public string Titulo;
    public string Autor;

    public Libro()
    {
        Titulo = "Sin título";
```

```
        Autor = "Anónimo";
    }

    public Libro(string titulo)
    {
        Titulo = titulo;
        Autor = "Anónimo";
    }

    public Libro(string titulo, string autor)
    {
        Titulo = titulo;
        Autor = autor;
    }
}
```

---

d. Constructor estático

Se ejecuta **una vez por clase**, no por objeto. Se usa para inicializar datos estáticos.

```
public class Configuracion
{
    public static string Ruta;

    // Constructor estático
    static Configuracion()
    {
        Ruta = "C:\\datos";
    }
}
```

---

3. ¿Qué es un Destructor?

Un **destructor** es un método especial que se ejecuta **automáticamente cuando el objeto es destruido** por el recolector de basura (GC).

🔧 Sintaxis:

```
~NombreClase()
{
    // Código para liberar recursos
```

```
}
```

¿Para qué se usa un destructor?

- Liberar recursos no administrados (conexiones, archivos, etc.).
- Hacer limpieza antes de que el objeto desaparezca.

⚠ Importante:

- C# **usa recolección de basura (GC)**, por eso **los destructores casi nunca se usan directamente**.
- Mejor usar `IDisposable` y el método `Dispose()` para manejar recursos explícitamente.

---

4. Ejemplo completo

```
using System;

public class Persona
{
    public string Nombre;
    public int Edad;

    // Constructor por defecto
    public Persona()
    {
        Nombre = "Desconocido";
        Edad = 0;
        Console.WriteLine("Se ha creado una persona.");
    }

    // Constructor parametrizado
    public Persona(string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
        Console.WriteLine($"Se ha creado a {Nombre}, de {Edad} años.");
    }
```

```
    // Destructor
    ~Persona()
    {
        Console.WriteLine($"La persona {Nombre} está siendo destruida.");
    }
}

class Program
{
    static void Main()
    {
        Persona p1 = new Persona();
        Persona p2 = new Persona("Ana", 25);
    }
}
```

---

5. Ejercicios propuestos

Ejercicio 1:

Crea una clase llamada `Animal` que tenga un constructor por defecto que diga "Animal creado", y otro parametrizado que reciba nombre y especie. Agrega un destructor que imprima "Animal destruido".

Ejercicio 2:

Crea una clase llamada `CuentaBancaria` con atributos `titular` y `saldo`. Crea dos constructores: uno por defecto y otro con parámetros. El constructor debe mostrar el estado inicial de la cuenta.

Ejercicio 3:

Implementa una clase `Servidor` con un **constructor estático** que configure la IP por defecto (`"127.0.0.1"`) y un método que muestre la IP actual. Usa el constructor estático para inicializar la IP solo una vez.

---

6. Soluciones

✔ Solución 1

```
public class Animal
{
```

```
    public string Nombre;
    public string Especie;

    public Animal()
    {
        Console.WriteLine("Animal creado");
    }

    public Animal(string nombre, string especie)
    {
        Nombre = nombre;
        Especie = especie;
        Console.WriteLine($"Animal creado: {Nombre}, {Especie}");
    }

    ~Animal()
    {
        Console.WriteLine("Animal destruido");
    }
}
```

✔ Solución 2

```
public class CuentaBancaria
{
    public string Titular;
    public double Saldo;

    public CuentaBancaria()
    {
        Titular = "Sin titular";
        Saldo = 0;
        Console.WriteLine($"Cuenta creada: {Titular}, Saldo: {Saldo}");
    }

    public CuentaBancaria(string titular, double saldo)
    {
```

```
        Titular = titular;
        Saldo = saldo;
        Console.WriteLine($"Cuenta creada: {Titular}, Saldo: {Saldo}");
    }
}
```

✔ Solución 3

```
public class Servidor
{
    public static string IP;

    static Servidor()
    {
        IP = "127.0.0.1";
        Console.WriteLine("Servidor configurado con IP por defecto");
    }

    public static void MostrarIP()
    {
        Console.WriteLine("IP actual: " + IP);
    }
}
```

🧾 7. Conclusión

Los constructores son herramientas fundamentales para crear objetos de forma organizada y segura. Los destructores, aunque menos usados, permiten liberar recursos cuando el objeto deja de existir.

Para recursos más críticos o sistemas que necesitan limpieza manual, se recomienda implementar `IDisposable` en lugar de usar destructores.

Lección 11: Modificadores de Acceso en C#

Objetivos

- Comprender qué son los modificadores de acceso y por qué se usan.
- Conocer todos los modificadores disponibles en C#.
- Aprender a aplicar el encapsulamiento correctamente.
- Practicar con ejercicios aplicados.

---

1. ¿Qué son los modificadores de acceso?

Los **modificadores de acceso** determinan **quién puede acceder a un miembro (atributo, método, clase)** desde otras partes del programa.

Su objetivo principal es **controlar la visibilidad** y proteger los datos, aplicando el principio de **encapsulamiento** de la programación orientada a objetos.

---

2. Tipos de Modificadores de Acceso en C#

Modificador	Acceso desde la misma clase	Subclases	Mismo ensamblado	Desde otras clases
`private`	✅	❌	❌	❌
`protected`	✅	✅	❌	❌
`internal`	✅	✅	✅	❌
`protected internal`	✅	✅	✅	❌
`private protected`	✅	✅ (solo en el mismo ensamblado)	✅	❌
`public`	✅	✅	✅	✅

---

3. Explicación de cada modificador

`private`

El más restrictivo. Solo accesible dentro de la misma clase.

```
class Persona
{
    private string nombre;

    public void SetNombre(string valor)
```

```
    {
        nombre = valor;
    }
}
```

---

protected

Accesible desde la misma clase y sus subclases.

```
class Animal
{
    protected void Respirar()
    {
        Console.WriteLine("Respirando...");
    }
}

class Perro : Animal
{
    public void Accion()
    {
        Respirar(); // válido
    }
}
```

---

internal

Accesible solo dentro del mismo **ensamblado o proyecto**.

```
internal class Servicio
{
    internal void Ejecutar() => Console.WriteLine("Ejecutando...");
}
```

---

protected internal

Accesible desde **subclases** o **dentro del mismo ensamblado**.

```
class Base
{
    protected internal void Metodo() => Console.WriteLine("Método accesible");
}
```

`private protected` (desde C# 7.2)

Accesible desde la misma clase o subclases, pero **solo dentro del mismo ensamblado**.

```
class Base

{

    private protected void Metodo() => Console.WriteLine("Solo subclases internas");

}
```

`public`

Totalmente accesible desde cualquier parte.

```
public class Utilidad

{

    public void Mostrar() => Console.WriteLine("Visible desde cualquier clase");

}
```

---

4. Buenas prácticas

- Usa `private` por defecto y **expón solo lo necesario**.
- Usa `public` para **interfaz pública clara y limitada**.
- Evita hacer atributos públicos directamente; usa propiedades.
- Usa `protected` para permitir que las clases derivadas accedan a lo necesario.
- **Encapsula lógica interna** que no debe ser expuesta al exterior.

---

5. Errores comunes

Error	Explicación
Usar `public` para todo	Viola el principio de encapsulamiento.
Hacer atributos públicos en lugar de usar propiedades	Exposición directa puede dañar la integridad del objeto.
Confundir `protected internal` con `private protected`	Uno permite acceso amplio en subclases y proyectos, el otro es más restringido.

---

6. Ejemplo práctico completo

```
public class Empleado

{

    private string nombre;

    protected int edad;
```

```
    public string Puesto;

    public Empleado(string nombre, int edad, string puesto)
    {
        this.nombre = nombre;
        this.edad = edad;
        this.Puesto = puesto;
    }

    protected void MostrarEdad()
    {
        Console.WriteLine($"Edad: {edad}");
    }

    public void Presentarse()
    {
        Console.WriteLine($"Hola, soy {nombre} y trabajo como {Puesto}.");
    }
}

class Gerente : Empleado
{
    public Gerente(string nombre, int edad, string puesto)
        : base(nombre, edad, puesto) { }

    public void MostrarInfo()
    {
        MostrarEdad(); // Acceso permitido a método protected
    }
}
```

---

7. Ejercicios propuestos

Ejercicio 1

Crea una clase `Libro` con:

- Atributo `titulo` (privado)

- Atributo `autor` (protegido)
- Atributo `editorial` (público)
- Método público que imprima toda la información del libro.

Ejercicio 2

Crea una clase `Cuenta` con:

- Atributo `saldo` (private)
- Método público `Depositar` y `ConsultarSaldo`
- Demuestra que no se puede acceder a `saldo` directamente desde fuera.

Ejercicio 3

Crea una clase base `Vehiculo` con un método `EncenderMotor` (protected).
Luego crea una subclase `Coche` que use ese método. ¿Puedes llamarlo desde el `Main`?

---

8. Soluciones

✔ Solución Ejercicio 1

```
public class Libro
{
    private string titulo;
    protected string autor;
    public string editorial;

    public Libro(string titulo, string autor, string editorial)
    {
        this.titulo = titulo;
        this.autor = autor;
        this.editorial = editorial;
    }

    public void MostrarInfo()
    {
        Console.WriteLine($"Título: {titulo}, Autor: {autor}, Editorial:
{editorial}");
    }
```

```
}
```

Solución Ejercicio 2

```
public class Cuenta
{
    private double saldo;

    public Cuenta()
    {
        saldo = 0;
    }

    public void Depositar(double cantidad)
    {
        if (cantidad > 0)
            saldo += cantidad;
    }

    public void ConsultarSaldo()
    {
        Console.WriteLine($"Saldo actual: {saldo}");
    }
}
```

Solución Ejercicio 3

```
class Vehiculo
{
    protected void EncenderMotor()
    {
        Console.WriteLine("Motor encendido");
    }
}

class Coche : Vehiculo
{
    public void Arrancar()
```

```
    {
        EncenderMotor(); // Acceso permitido
    }
}

// En Main:
Coche miCoche = new Coche();
miCoche.Arrancar(); // OK
// miCoche.EncenderMotor(); ❌ Error: método es protected
```

---

9. Conclusión

Los modificadores de acceso son esenciales para mantener un diseño limpio, seguro y mantenible. Saber cuándo usar cada uno permite crear clases más robustas, evitando errores y haciendo tu código más profesional.

Lección 12: Excepciones y Manejo de Errores en C#

1. ¿Qué es una excepción?

Una **excepción** es un error que ocurre durante la ejecución de un programa. En C#, las excepciones son objetos derivados de la clase base `System.Exception`.

Ejemplos comunes de excepciones:

- `DivideByZeroException`
- `NullReferenceException`
- `IndexOutOfRangeException`
- `FileNotFoundException`
- `FormatException`

2. Bloques try, catch y finally

Sintaxis básica:

```
try
{
    // Código que puede lanzar una excepción
}
catch (ExceptionTipo ex)
{
    // Código para manejar la excepción
}
finally
{
    // Código que siempre se ejecuta (opcional)
}
```

Ejemplo:

```
try
{
    int x = 10;
    int y = 0;
    int resultado = x / y;
}
```

```
catch (DivideByZeroException ex)
{
    Console.WriteLine("Error: División por cero.");
}
finally
{
    Console.WriteLine("Este bloque siempre se ejecuta.");
}
```

3. Jerarquía de excepciones

```
System.Object
└── System.Exception
    ├── System.SystemException
    │   ├── System.ArithmeticException
    │   │   └── DivideByZeroException
    │   ├── IndexOutOfRangeException
    │   └── NullReferenceException
    └── System.IO.IOException
        └── FileNotFoundException
```

4. Múltiples bloques catch

Puedes capturar diferentes tipos de errores:

```
try
{
    string numero = "abc";
    int n = int.Parse(numero);
}
catch (FormatException)
{
    Console.WriteLine("Formato inválido.");
}
catch (Exception ex)
{
    Console.WriteLine("Error general: " + ex.Message);
}
```

5. La palabra clave `throw`

Permite lanzar manualmente una excepción.

Ejemplo:

```
throw new Exception("Error personalizado");
```

También se puede relanzar la excepción capturada:

```
catch (Exception ex)
{
    Console.WriteLine("Registrando error...");
    throw; // vuelve a lanzar la misma excepción
}
```

6. Crear excepciones personalizadas

Puedes definir tus propias clases de excepción:

```
class MiExcepcion : Exception
{
    public MiExcepcion(string mensaje) : base(mensaje) { }
}
```

Uso:

```
throw new MiExcepcion("Ocurrió un error personalizado.");
```

7. Uso del bloque `finally`

Se usa para liberar recursos, cerrar archivos, conexiones, etc.

```
FileStream archivo = null;
try
{
    archivo = new FileStream("datos.txt", FileMode.Open);
    // Operaciones con archivo
}
catch (IOException ex)
{
    Console.WriteLine("Archivo no encontrado.");
}
finally
{
    if (archivo != null)
        archivo.Close();
```

```
}
```

8. Buenas prácticas

- **Evita capturar excepciones genéricas** salvo para registrar errores inesperados.
- **Lanza excepciones solo cuando sea necesario.**
- **Utiliza `finally` para liberar recursos.**
- **No uses excepciones para control de flujo.**

9. Ejercicio práctico

Crea una calculadora básica que permita sumar, restar, multiplicar y dividir dos números. Debe manejar:

- división por cero
- errores de formato
- operaciones inválidas

```
using System;

class Calculadora
{
    static void Main()
    {
        try
        {
            Console.Write("Ingrese el primer número: ");
            double a = double.Parse(Console.ReadLine());

            Console.Write("Ingrese el segundo número: ");
            double b = double.Parse(Console.ReadLine());

            Console.Write("Ingrese la operación (+, -, *, /): ");
            string op = Console.ReadLine();

            double resultado = op switch
            {
                "+" => a + b,
                "-" => a - b,
                "*" => a * b,
```

```
                "/" => b != 0 ? a / b : throw new DivideByZeroException(),
                _ => throw new InvalidOperationException("Operación no válida.")
            };

            Console.WriteLine("Resultado: " + resultado);
        }
        catch (FormatException)
        {
            Console.WriteLine("Entrada no válida. Por favor, introduzca
números.");
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("No se puede dividir entre cero.");
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error inesperado: " + ex.Message);
        }
        finally
        {
            Console.WriteLine("Fin del programa.");
        }
    }
}
```

Parte III – Programación Avanzada

Lección Completa 13: Delegados y Eventos en C#

En C#, los delegados y eventos son herramientas fundamentales para construir aplicaciones basadas en eventos, desacoplar clases y manejar llamadas a métodos en tiempo de ejecución.

Objetivos

Al finalizar esta lección, podrás:

- Entender qué es un delegado y para qué se utiliza.
- Crear y utilizar delegados personalizados.
- Entender qué es un evento y cómo se relaciona con los delegados.
- Declarar y manejar eventos en tus propias clases.
- Comprender cómo el patrón de publicación-suscripción funciona en C#.

---

1. ¿Qué es un Delegado?

Un **delegado** es un tipo que representa una referencia a uno o varios métodos con una firma determinada.

Sintaxis básica:

```
delegate void MiDelegado(string mensaje);
```

Este delegado puede apuntar a cualquier método que reciba un `string` y devuelva `void`.

Ejemplo básico:

```
using System;


delegate void SaludoDelegate(string nombre);


class Program
```

```
{
    static void Saludar(string nombre)
    {
        Console.WriteLine($"Hola, {nombre}!");
    }

    static void Main()
    {
        SaludoDelegate delegado = new SaludoDelegate(Saludar);
        delegado("María");
    }
}
```

Salida:

```
Hola, María!
```

---

2. Delegados Multicast

Un delegado puede contener una **lista de métodos**.

```
using System;

delegate void Operacion();

class Program
{
    static void Metodo1() => Console.WriteLine("Método 1 ejecutado");
    static void Metodo2() => Console.WriteLine("Método 2 ejecutado");

    static void Main()
    {
        Operacion delegado = Metodo1;
        delegado += Metodo2;

        delegado(); // Invoca ambos métodos
    }
}
```

---

3. Delegados Genéricos: Action, Func y Predicate

Action<T>

Un delegado que **no devuelve valor** (void):

```
Action<string> saludar = nombre => Console.WriteLine($"Hola, {nombre}");
saludar("Ana");
```

Func<T, TResult>

Devuelve un resultado:

```
Func<int, int, int> sumar = (a, b) => a + b;
Console.WriteLine(sumar(3, 4));  // 7
```

Predicate<T>

Devuelve true o false:

```
Predicate<int> esPar = n => n % 2 == 0;
Console.WriteLine(esPar(4));  // True
```

---

4. ¿Qué es un Evento?

Un **evento** es una notificación de que algo ha ocurrido. Se declara utilizando un delegado.

Sintaxis:

```
public event MiDelegado MiEvento;
```

Ejemplo completo de evento:

```
using System;

delegate void Notificar();

class Emisor
{
    public event Notificar Alerta;

    public void ActivarAlerta()
    {
        Console.WriteLine("Activando alerta...");
        Alerta?.Invoke();
    }
}
```

```
class Receptor
{
    public void MetodoRespuesta()
    {
        Console.WriteLine("¡Alerta recibida!");
    }
}

class Program
{
    static void Main()
    {
        Emisor emisor = new Emisor();
        Receptor receptor = new Receptor();

        emisor.Alerta += receptor.MetodoRespuesta;
        emisor.ActivarAlerta();
    }
}
```

---

5. Evento con argumentos (EventHandler y EventArgs)

Usamos `EventHandler` para pasar información adicional.

Ejemplo con `EventHandler`:

```
using System;

class Sensor
{
    public event EventHandler TemperaturaAlta;

    public void Medir(int temperatura)
    {
        if (temperatura > 30)
        {
            TemperaturaAlta?.Invoke(this, EventArgs.Empty);
```

```
        }
    }
}

class Alarma
{
    public void ActivarAlarma(object sender, EventArgs e)
    {
        Console.WriteLine("¡Alarma: temperatura elevada!");
    }
}

class Program
{
    static void Main()
    {
        Sensor sensor = new Sensor();
        Alarma alarma = new Alarma();

        sensor.TemperaturaAlta += alarma.ActivarAlarma;

        sensor.Medir(35); // Dispara evento
    }
}
```

---

6. Patrón Publicador-Suscriptor

- El **publicador** es la clase que define el evento.
- El **suscriptor** es la clase que maneja el evento.

Este patrón promueve el **desacoplamiento** entre componentes.

---

7. Buenas prácticas

- Usa `EventHandler` o `EventHandler<T>` para eventos.
- Protege el evento con `?.Invoke()` para evitar errores si no hay suscriptores.
- Usa nombres significativos para eventos como `OnDatosActualizados`.

Ejercicios Propuestos

Ejercicio 1:

Crea un programa que use un delegado llamado `OperacionBinaria` para sumar, restar y multiplicar dos números.

Ejercicio 2:

Crea una clase `Reloj` que lance un evento `AlPasarUnSegundo`. Suscríbete al evento desde otra clase que muestre “Un segundo ha pasado”.

Ejercicio 3:

Crea una clase `Boton` con un evento `AlHacerClick`. Al presionar enter desde consola, dispara el evento.

Soluciones

Ejercicio 1 - Solución

```
using System;

delegate int OperacionBinaria(int a, int b);

class Program
{
    static int Sumar(int a, int b) => a + b;
    static int Restar(int a, int b) => a - b;
    static int Multiplicar(int a, int b) => a * b;

    static void Main()
    {
        OperacionBinaria op = Sumar;
        Console.WriteLine("Suma: " + op(3, 4));

        op = Restar;
        Console.WriteLine("Resta: " + op(7, 2));

        op = Multiplicar;
        Console.WriteLine("Multiplicación: " + op(5, 5));
    }
```

```
}
```

Ejercicio 2 - Solución

```
using System;
using System.Threading;

class Reloj
{
    public event EventHandler AlPasarUnSegundo;

    public void Iniciar()
    {
        while (true)
        {
            Thread.Sleep(1000);
            AlPasarUnSegundo?.Invoke(this, EventArgs.Empty);
        }
    }
}

class Oyente
{
    public void MostrarMensaje(object sender, EventArgs e)
    {
        Console.WriteLine("Un segundo ha pasado.");
    }
}

class Program
{
    static void Main()
    {
        Reloj reloj = new Reloj();
        Oyente oyente = new Oyente();

        reloj.AlPasarUnSegundo += oyente.MostrarMensaje;
```

```
        reloj.Iniciar();
    }
}
```

Ejercicio 3 - Solución

```
using System;

class Boton
{
    public event EventHandler AlHacerClick;

    public void EsperarClick()
    {
        Console.WriteLine("Pulsa ENTER para hacer clic...");
        Console.ReadLine();
        AlHacerClick?.Invoke(this, EventArgs.Empty);
    }
}

class Programa
{
    static void Main()
    {
        Boton boton = new Boton();
        boton.AlHacerClick += (s, e) => Console.WriteLine("¡Botón presionado!");

        boton.EsperarClick();
    }
}
```

Conclusión

Los delegados y eventos son herramientas poderosas para hacer tu código más **modular**, **escalable** y **desacoplado**. Son esenciales para interfaces gráficas, controladores de eventos y diseño de sistemas complejos.

Lección 14: Expresiones Lambda y LINQ en C#

Objetivos de la lección

Al terminar esta lección podrás:

- Entender qué es una **expresión lambda** y cómo usarla.
- Utilizar **funciones anónimas** para simplificar tu código.
- Entender los conceptos clave de **LINQ** (Language Integrated Query).
- Consultar y transformar colecciones usando **LINQ con expresiones lambda y sintaxis de consulta**.
- Resolver problemas reales utilizando ambas herramientas de forma combinada.

---

1. ¿Qué es una Expresión Lambda?

Una **expresión lambda** es una forma concisa de escribir funciones anónimas.

Sintaxis general:

```
(parametros) => expresión_o_bloque
```

Ejemplos básicos:

```
Func<int, int> cuadrado = x => x * x;
Console.WriteLine(cuadrado(5));  // 25

Action<string> saludar = nombre => Console.WriteLine($"Hola, {nombre}!");
saludar("Ana");
```

¿Dónde se usan las expresiones lambda?

- Con delegados (`Func`, `Action`, `Predicate`)
- Con LINQ
- Con eventos
- Con métodos que aceptan funciones como parámetro (`List<T>.Find()`, `Where()`, etc.)

---

2. Delegados integrados de C#

- `Func<T, TResult>`: devuelve un valor.

- `Action<T>`: no devuelve valor.
- `Predicate<T>`: devuelve un bool.

Ejemplo:

```
List<int> numeros = new List<int> { 1, 2, 3, 4, 5 };
List<int> pares = numeros.FindAll(x => x % 2 == 0);

pares.ForEach(n => Console.WriteLine(n));  // 2, 4
```

---

3. ¿Qué es LINQ?

LINQ (Language Integrated Query) es un conjunto de características de C# que permite **consultar datos** de forma declarativa.

LINQ puede trabajar con:

- Arrays
- Listas
- Bases de datos (LINQ to SQL, Entity Framework)
- XML
- Archivos

Sintaxis de Consulta vs Sintaxis de Método

📃 Sintaxis de Consulta:

```
var resultado = from n in numeros
                where n > 3
                select n;
```

📃 Sintaxis de Método (con lambdas):

```
var resultado = numeros.Where(n => n > 3);
```

Ambas hacen lo mismo. La sintaxis de método es más común en programación moderna.

---

4. Métodos LINQ comunes

Método	Descripción
`Where`	Filtra elementos
`Select`	Proyecta datos
`OrderBy`, `OrderByDescending`	Ordena
`GroupBy`	Agrupa
`First`, `FirstOrDefault`	Primer elemento

Método	Descripción
`Any`, `All`	Condiciones lógicas
`Sum`, `Average`, `Count`	Operaciones numéricas
`ToList`, `ToArray`	Conversión

Ejemplos prácticos

```
List<int> numeros = new List<int> { 1, 2, 3, 4, 5, 6 };

var pares = numeros.Where(n => n % 2 == 0);         // Filtrar pares
var dobles = numeros.Select(n => n * 2);            // Duplicar cada número
var suma = numeros.Sum();                           // Sumar
var hayImpares = numeros.Any(n => n % 2 != 0);      // ¿Hay impares?

foreach (var n in pares)
    Console.WriteLine(n);  // 2, 4, 6
```

---

5. Uso avanzado con objetos

```
class Persona
{
    public string Nombre { get; set; }
    public int Edad { get; set; }
}

List<Persona> personas = new List<Persona>
{
    new Persona { Nombre = "Ana", Edad = 28 },
    new Persona { Nombre = "Luis", Edad = 35 },
    new Persona { Nombre = "Eva", Edad = 22 }
};

// Obtener nombres de mayores de 25
var mayores = personas
              .Where(p => p.Edad > 25)
              .Select(p => p.Nombre);

foreach (var nombre in mayores)
```

```
    Console.WriteLine(nombre);  // Ana, Luis
```

---

6. Agrupaciones y ordenamientos

```
var agrupadoPorEdad = personas.GroupBy(p => p.Edad);

foreach (var grupo in agrupadoPorEdad)
{
    Console.WriteLine($"Edad: {grupo.Key}");
    foreach (var persona in grupo)
        Console.WriteLine($" - {persona.Nombre}");
}
```

---

7. Encadenamiento de métodos LINQ

```
var resultado = personas
    .Where(p => p.Edad > 20)
    .OrderBy(p => p.Nombre)
    .Select(p => p.Nombre.ToUpper());

foreach (var nombre in resultado)
    Console.WriteLine(nombre);
```

Ejercicios propuestos

✍ Ejercicio 1:

Crea una lista de enteros. Filtra los números impares, multiplícalos por 3 y ordénalos de mayor a menor.

✍ Ejercicio 2:

Crea una lista de personas con nombre y edad. Filtra los menores de 30, y muestra sus nombres en orden alfabético.

✍ Ejercicio 3:

Crea una lista de productos (nombre, precio). Filtra los que cuesten más de 50, y muestra su nombre y precio con IVA (21%).

Soluciones

Ejercicio 1:

```
List<int> numeros = new List<int> { 1, 4, 7, 9, 10, 12 };

var resultado = numeros
    .Where(n => n % 2 != 0)
    .Select(n => n * 3)
    .OrderByDescending(n => n);

foreach (var n in resultado)
    Console.WriteLine(n);  // 27, 21, 3
```

Ejercicio 2:

```
class Persona
{
    public string Nombre { get; set; }
    public int Edad { get; set; }
}

List<Persona> personas = new List<Persona>
{
    new Persona { Nombre = "Carlos", Edad = 25 },
    new Persona { Nombre = "Ana", Edad = 32 },
    new Persona { Nombre = "Bea", Edad = 22 }
};

var menores30 = personas
    .Where(p => p.Edad < 30)
    .OrderBy(p => p.Nombre)
    .Select(p => p.Nombre);

foreach (var nombre in menores30)
    Console.WriteLine(nombre);  // Bea, Carlos
```

Ejercicio 3:

```
class Producto
{
    public string Nombre { get; set; }
    public double Precio { get; set; }
}

List<Producto> productos = new List<Producto>
{
    new Producto { Nombre = "Teclado", Precio = 30 },
    new Producto { Nombre = "Pantalla", Precio = 120 },
    new Producto { Nombre = "Ratón", Precio = 20 }
};

var conIva = productos
    .Where(p => p.Precio > 50)
    .Select(p => new { p.Nombre, PrecioConIva = p.Precio * 1.21 });

foreach (var p in conIva)
    Console.WriteLine($"{p.Nombre}: {p.PrecioConIva:F2} €");
```

Conclusión

Las **expresiones lambda** y **LINQ** son herramientas poderosas que te permiten escribir código más limpio, expresivo y funcional. Son esenciales para el desarrollo moderno en C#.

Lección 15: Programación Asíncrona en C#

Objetivos de la lección

- Comprender qué es la programación asíncrona y por qué es importante.
- Usar correctamente async, await, Task y Task<T>.
- Aprender a evitar bloqueos en la interfaz de usuario y mejorar el rendimiento.
- Aplicar técnicas de asincronía en aplicaciones de consola, escritorio y web.
- Detectar y evitar errores comunes en programación asíncrona.

---

1. Introducción a la Programación Asíncrona

¿Qué es la programación asíncrona?

La programación asíncrona permite ejecutar tareas de larga duración (como llamadas a archivos, redes o bases de datos) sin bloquear el hilo principal de ejecución.

¿Por qué usarla?

- Mejora la **experiencia del usuario** (la aplicación no se congela).
- Permite **mejor rendimiento** y **respuesta rápida**.
- Utiliza mejor los **recursos del sistema**.

---

2. Palabras clave y conceptos básicos

Task y Task<T>

- Representan una operación asincrónica.
- Task no devuelve valor.
- Task<T> devuelve un resultado de tipo T.

async y await

- async: marca un método que contiene código asincrónico.
- await: espera sin bloquear a que la tarea termine.

```
public async Task DescargarDatosAsync()
{
    string datos = await ObtenerDatosDesdeInternetAsync();
    Console.WriteLine(datos);
}
```

---

3. Ejemplo básico

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        Console.WriteLine("Descargando datos...");
        string resultado = await DescargarDatosAsync();
        Console.WriteLine("Resultado:");
        Console.WriteLine(resultado);
    }

    static async Task<string> DescargarDatosAsync()
    {
        using HttpClient client = new HttpClient();
        string contenido = await client.GetStringAsync("https://example.com");
        return contenido;
    }
}
```

---

4. Aplicaciones prácticas

a) Leer un archivo de forma asíncrona

```
using System.IO;
using System.Threading.Tasks;

public async Task<string> LeerArchivoAsync(string ruta)
{
    using StreamReader lector = new StreamReader(ruta);
    return await lector.ReadToEndAsync();
}
```

b) Llamadas a API REST

```
using System.Net.Http;


public async Task<string> ObtenerJsonDesdeApiAsync()
{
    using HttpClient client = new HttpClient();
    HttpResponseMessage respuesta = await
client.GetAsync("https://api.github.com");
    return await respuesta.Content.ReadAsStringAsync();
}
```

⚠ 5. Buenas prácticas

- Usa `ConfigureAwait(false)` en bibliotecas para evitar dependencias del contexto de sincronización.
- Evita `.Result` o `.Wait()` porque pueden causar **deadlocks**.
- Usa `try/catch` para capturar excepciones asíncronas.
- Usa `CancellationToken` para cancelar tareas si es necesario.

---

6. Errores comunes

Error	Explicación	Ejemplo
Bloqueo con `.Result`	Causa deadlock en UI	`var r = ObtenerDatos().Result;`
No esperar con `await`	No se ejecuta como se espera	`Guardar().Wait();`
No usar `async` en eventos	Se pierde el control del flujo	`button.Click += (s,e) => Descargar();`

---

7. Ejercicio guiado paso a paso

Enunciado:

Crear una aplicación que descargue el contenido de una web y lo guarde en un archivo **de forma asíncrona**.

Solución:

```
using System;
using System.IO;
using System.Net.Http;
using System.Threading.Tasks;
```

```
class Program
{
    static async Task Main()
    {
        Console.WriteLine("Descargando contenido...");
        string contenido = await DescargarContenidoAsync("https://example.com");

        Console.WriteLine("Guardando en archivo...");
        await GuardarEnArchivoAsync("contenido.txt", contenido);

        Console.WriteLine("¡Hecho!");
    }

    static async Task<string> DescargarContenidoAsync(string url)
    {
        using HttpClient client = new HttpClient();
        return await client.GetStringAsync(url);
    }

    static async Task GuardarEnArchivoAsync(string ruta, string contenido)
    {
        await File.WriteAllTextAsync(ruta, contenido);
    }
}
```

---

8. Ejercicios prácticos

1. Realiza una suma de dos números, pero simula que tarda 2 segundos (usa `Task.Delay`).

2. Escribe una función que lea varias líneas de un archivo usando `StreamReader.ReadLineAsync()`.

3. Implementa una descarga simultánea de tres páginas web y muestra sus longitudes.

4. Modifica un botón de Windows Forms para que su evento `Click` descargue un archivo asíncronamente sin bloquear la interfaz.

---

9. Herramientas útiles

- `Task.WhenAll()`: espera varias tareas.
- `Task.Run()`: ejecuta código en un hilo separado.
- `SemaphoreSlim`: controla acceso simultáneo en tareas.
- `CancellationToken`: permite cancelar operaciones.

---

10. Recursos recomendados

- Documentación oficial de Microsoft: Asynchronous Programming in C#
- Libro: *Asynchronous Programming with C# and .NET* (Microsoft Press)
- Curso gratuito en YouTube: "C# async/await Deep Dive"

Conclusión

La programación asíncrona en C# es esencial para crear aplicaciones modernas, eficientes y responsivas. Dominar `async/await` te permite trabajar con procesos largos sin bloquear la ejecución, mejorando notablemente la experiencia del usuario.

Lección 16: Programación Funcional en C#

Objetivos

Al final de esta lección podrás:

- Comprender qué es la programación funcional y cómo se aplica en C#.
- Usar funciones puras, expresiones lambda, funciones de orden superior y LINQ.
- Aplicar principios funcionales como inmutabilidad y composición.
- Conocer herramientas del framework como `Func`, `Action`, `Predicate`.

---

1. ¿Qué es la Programación Funcional?

La programación funcional es un paradigma donde la computación se basa en la evaluación de funciones matemáticas, evitando estados mutables y efectos secundarios.

Características clave:

- Funciones puras
- Inmutabilidad
- Declarativo vs Imperativo
- Composición de funciones
- Funciones de orden superior

> C# no es puramente funcional, pero **soporta** programación funcional desde C# 3.0 (con LINQ y lambdas) y se ha potenciado aún más desde C# 6 en adelante.

---

2. Funciones Puras

Una **función pura** es aquella que:

1. Siempre devuelve el mismo resultado para los mismos argumentos.
2. No tiene efectos secundarios (no modifica nada fuera de sí).

Ejemplo:

```
int Sumar(int a, int b)
{
    return a + b; // función pura
}
```

Ejemplo no puro:

```
int contador = 0;
```

```
int Incrementar(int x)
{
    contador++; // efecto secundario
    return x + contador;
}
```

---

3. Funciones de Orden Superior

Son funciones que:

- Aceptan funciones como parámetros.
- Devuelven funciones como resultado.

Ejemplo con `Func<T>`:

```
Func<int, int, int> suma = (a, b) => a + b;
Console.WriteLine(suma(3, 4)); // 7
```

Función que recibe otra función:

```
void EjecutarOperacion(Func<int, int, int> operacion, int a, int b)
{
    Console.WriteLine(operacion(a, b));
}

EjecutarOperacion((x, y) => x * y, 4, 5); // 20
```

---

4. Expresiones Lambda

C# permite definir funciones de forma concisa con **lambdas**:

```
// (parámetros) => expresión
Func<int, int, int> resta = (a, b) => a - b;
```

Las lambdas son la base de muchas operaciones funcionales, como en LINQ o delegados.

---

5. Delegados Funcionales: Func, Action, Predicate

- `Func<T, TResult>`: devuelve un valor.
- `Action<T>`: no devuelve valor.
- `Predicate<T>`: devuelve `bool`.

Ejemplos:

```
Func<int, int> cuadrado = x => x * x;
```

```
Action<string> saludar = nombre => Console.WriteLine($"Hola, {nombre}");
Predicate<int> esPar = x => x % 2 == 0;
```

---

6. Inmutabilidad

La programación funcional favorece estructuras de datos **inmutables**.

En C#:

- Se puede usar `readonly`, `record`, o simplemente no modificar las variables.

Ejemplo:

```
record Persona(string Nombre, int Edad); // inmutable

var p1 = new Persona("Ana", 30);
var p2 = p1 with { Edad = 31 }; // no modifica p1
```

---

7. Composición de funciones

La composición consiste en encadenar funciones, donde la salida de una es la entrada de otra.

Ejemplo:

```
Func<int, int> doble = x => x * 2;
Func<int, int> sumarTres = x => x + 3;

// Composición manual
Func<int, int> compuesto = x => sumarTres(doble(x));
Console.WriteLine(compuesto(5)); // (5*2)+3 = 13
```

---

8. LINQ: Programación funcional sobre colecciones

LINQ permite operar sobre colecciones de forma declarativa y funcional.

Ejemplo básico:

```
int[] numeros = { 1, 2, 3, 4, 5 };

var pares = numeros
    .Where(n => n % 2 == 0)
    .Select(n => n * 10);

foreach (var x in pares)
    Console.WriteLine(x); // 20, 40
```

Funciones más comunes:

- `Where` → filtra
- `Select` → transforma
- `Aggregate` → reduce
- `Any`, `All`, `First`, `Count`

---

9. Ejercicio práctico: Funcional vs Imperativo

Imperativo:

```
int suma = 0;
foreach (var n in numeros)
    if (n % 2 == 0)
        suma += n * 10;
```

Funcional:

```
var suma = numeros
    .Where(n => n % 2 == 0)
    .Select(n => n * 10)
    .Sum();
```

---

10. Prácticas recomendadas

- Prefiere funciones puras.
- Usa LINQ para manipular colecciones.
- Evita estados mutables globales.
- Usa `Func`, `Action`, `Predicate` en lugar de delegados personalizados.
- Aprovecha `record` y `with` para trabajar con datos inmutables.

---

11. Mini proyecto: Calculadora funcional de descuentos

```
using System;

class Program
{
    static void Main()
    {
        Func<decimal, decimal> descuento10 = precio => precio * 0.9M;
```

```
        Func<decimal, decimal> impuestoIVA = precio => precio * 1.21M;


        Func<decimal, decimal> precioFinal = precio =>
impuestoIVA(descuento10(precio));


        decimal resultado = precioFinal(100);
        Console.WriteLine($"Precio final: {resultado:F2}"); // 108.90
    }
}
```

Resumen

Concepto	Descripción
Funciones puras	Sin efectos secundarios
Func	Devuelve resultado
Action	No devuelve resultado
Predicate	Devuelve `bool`
Lambdas	Funciones anónimas
Inmutabilidad	No alterar estado
Composición	Unir funciones
LINQ	Consulta funcional sobre datos

Ejercicios recomendados

1. Crea una función pura que convierta una lista de temperaturas Celsius a Fahrenheit.
2. Usa LINQ para filtrar nombres que empiecen por vocal.
3. Composición: crea una función que eleve un número al cuadrado y luego reste 5.
4. Define un `record` para representar libros y filtra por año de publicación usando LINQ.
5. Implementa una función que reciba otra función como parámetro y la ejecute sobre un arreglo.

Lección 17: Generics en C#

Objetivos de Aprendizaje

Al final de esta lección, serás capaz de:

- Comprender qué son los *Generics* en C#.
- Saber por qué se utilizan y sus ventajas.
- Crear clases, métodos, interfaces y estructuras genéricas.
- Usar restricciones (*constraints*) en *Generics*.
- Comprender cómo funcionan las colecciones genéricas como `List<T>`, `Dictionary<TKey, TValue>`, etc.

---

1. ¿Qué son los *Generics*?

Generics (genéricos) permiten definir clases, métodos, interfaces y estructuras con un tipo de dato como parámetro. Esto aumenta la **reutilización del código**, **seguridad en tiempo de compilación** y **eficiencia de rendimiento**.

Ejemplo sin Generics

```
public class CajaEntero {
    private int valor;
    public void Guardar(int v) {
        valor = v;
    }
    public int Obtener() {
        return valor;
    }
}
```

Mismo ejemplo con Generics

```
public class Caja<T> {
    private T valor;
    public void Guardar(T v) {
        valor = v;
    }
    public T Obtener() {
        return valor;
```

```
    }
}
```

---

2. ¿Por qué usar *Generics*?

Ventajas:

- **Reutilización**: evita la duplicación de código para distintos tipos.
- **Seguridad**: errores en tiempo de compilación.
- **Rendimiento**: evita *boxing/unboxing* con tipos primitivos.

---

3. Sintaxis Básica

```
class NombreClase<T> {
    public T Dato { get; set; }
}
```

- `T` es un **parámetro de tipo**, puede ser cualquier identificador (`T`, `U`, `K`, etc.)
- Puedes usar múltiples parámetros: `class MiClase<T, U>`

---

4. Ejemplo Práctico de Clase Genérica

```
public class Caja<T> {
    private T contenido;
    public void Guardar(T valor) {
        contenido = valor;
    }
    public T Obtener() {
        return contenido;
    }
}

class Program {
    static void Main() {
        Caja<string> cajaDeTexto = new Caja<string>();
        cajaDeTexto.Guardar("Hola Mundo");
        Console.WriteLine(cajaDeTexto.Obtener());

        Caja<int> cajaDeNumeros = new Caja<int>();
```

```
        cajaDeNumeros.Guardar(42);
        Console.WriteLine(cajaDeNumeros.Obtener());
    }
}
```

---

5. Métodos Genéricos

Puedes crear métodos genéricos incluso en clases no genéricas:

```
public class Utilidades {
    public static void Intercambiar<T>(ref T a, ref T b) {
        T temp = a;
        a = b;
        b = temp;
    }
}
```

Uso:

```
int x = 5, y = 10;
Utilidades.Intercambiar(ref x, ref y);
Console.WriteLine($"{x}, {y}"); // 10, 5
```

---

6. Interfaces Genéricas

```
public interface IRepository<T> {
    void Add(T item);
    T Get(int id);
}
```

Implementación:

```
public class Repository<T> : IRepository<T> {
    private List<T> data = new List<T>();
    public void Add(T item) {
        data.Add(item);
    }

    public T Get(int id) {
        return data[id];
    }
}
```

7. Estructuras Genéricas

```
public struct Par<T1, T2> {
    public T1 Primero;
    public T2 Segundo;
}
```

Uso:

```
var par = new Par<int, string> { Primero = 1, Segundo = "uno" };
Console.WriteLine($"{par.Primero} - {par.Segundo}");
```

8. Restricciones de Tipo (Constraints)

Puedes restringir los tipos que se pueden usar como parámetros genéricos.

Tipos de restricciones:

Restricción	Significado
`where T : struct`	Solo tipos por valor
`where T : class`	Solo tipos por referencia
`where T : new()`	Debe tener constructor sin parámetros
`where T : BaseClass`	Hereda de clase base
`where T : Interface`	Implementa interfaz

Ejemplo:

```
public class Fabrica<T> where T : new() {
    public T Crear() {
        return new T();
    }
}
```

9. Colecciones Genéricas del .NET Framework

Clase	Descripción
`List<T>`	Lista dinámica de elementos
`Dictionary<K,V>`	Pares clave-valor
`Queue<T>`	Cola FIFO
`Stack<T>`	Pila LIFO
`HashSet<T>`	Conjunto sin duplicados

Ejemplo con `List<T>`

```
List<string> frutas = new List<string>();
frutas.Add("Manzana");
```

```
frutas.Add("Banana");

foreach (var fruta in frutas)
    Console.WriteLine(fruta);
```

---

10. Buenas Prácticas

- Usa nombres significativos para los parámetros genéricos (`T`, `TItem`, `TEntity`, etc.).
- Usa restricciones para evitar errores en tiempo de ejecución.
- Evita abusar de la generalización: no todo debe ser genérico.

---

11. Ejercicio Propuesto

Crea una clase genérica `Almacen<T>` que:

- Permita guardar hasta 5 elementos de tipo `T`.
- Tenga un método `AgregarElemento(T e)` y `ObtenerElemento(int i)`.

Sugerencia de solución:

```
public class Almacen<T> {
    private T[] elementos = new T[5];
    private int contador = 0;

    public void AgregarElemento(T e) {
        if (contador < 5) {
            elementos[contador] = e;
            contador++;
        } else {
            Console.WriteLine("Almacén lleno");
        }
    }

    public T ObtenerElemento(int i) {
        if (i >= 0 && i < contador) {
            return elementos[i];
        }
        throw new IndexOutOfRangeException("Índice inválido");
    }
```

```
}
```

---

12. Práctica Avanzada

Implementa un repositorio genérico `Repositorio<T>` con operaciones:

- `Agregar(T item)`
- `Eliminar(T item)`
- `Buscar(Func<T, bool> criterio)`
- `Listar()`

---

13. Preguntas de Autoevaluación

1. ¿Qué ventajas tienen los *Generics* frente a clases sin tipo?
2. ¿Qué diferencia hay entre `class MiClase<T>` y `void Metodo<T>()`?
3. ¿Qué significa la restricción `where T : new()`?
4. ¿Qué pasa si intentas usar un tipo de valor con una clase genérica restringida a `class`?
5. ¿Puedes usar múltiples restricciones en un mismo parámetro? ¿Cómo?

---

14. Conclusión

Los *Generics* en C# son una poderosa herramienta que permite crear código flexible, reutilizable, seguro y eficiente. Su uso se extiende desde estructuras de datos hasta servicios complejos de almacenamiento, inyección de dependencias y mucho más.

Parte IV – Interfaces Gráficas con Windows Forms

Lección 18: Introducción a Windows Forms con C#

1. ¿Qué es Windows Forms?

Windows Forms es una parte de la biblioteca de clases de .NET que permite crear interfaces gráficas de usuario (GUI) para aplicaciones de escritorio en sistemas Windows. Ofrece un conjunto de controles visuales (botones, cajas de texto, etiquetas, etc.) que permiten al usuario interactuar con la aplicación.

Es ideal para:

- Aplicaciones empresariales.
- Formularios de entrada de datos.
- Herramientas administrativas de escritorio.

---

2. Requisitos previos

- Conocimientos básicos de C# (variables, clases, métodos, eventos).
- Tener instalado **Visual Studio** con soporte para .NET.

---

3. Crear tu primer proyecto Windows Forms

Paso 1: Crear el proyecto

1. Abrir Visual Studio.
2. Seleccionar **Crear nuevo proyecto**.
3. Elegir **Aplicación de Windows Forms (.NET)**.
4. Asignar un nombre (ej. `MiPrimeraAppWinForms`).
5. Elegir la carpeta de destino y pulsar "Crear".

Paso 2: Conocer el entorno

- **Diseñador de formularios**: arrastrar y soltar controles visuales.
- **Propiedades**: modificar apariencia y comportamiento de los controles.
- **Explorador de soluciones**: ver archivos y componentes del proyecto.
- **Editor de código**: escribir el comportamiento (lógica) de la aplicación.

---

4. Estructura básica de un Formulario

Cuando creas una aplicación, se genera un formulario por defecto llamado `Form1.cs`.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

- `Form1` hereda de `Form`, que representa una ventana.
- `InitializeComponent()` inicializa todos los controles del formulario.

---

5. Agregar controles básicos

Ejemplo: Agregar controles desde el diseñador

1. Arrastrar un **Label**, **TextBox** y un **Button** desde la caja de herramientas.
2. Cambiar los nombres en la ventana **Propiedades**:
    - Label → `labelSaludo`
    - TextBox → `textBoxNombre`
    - Button → `buttonSaludar`

Ejemplo: Código para responder al botón

Haz doble clic en el botón para crear un evento `Click`:

```
private void buttonSaludar_Click(object sender, EventArgs e)
{
    string nombre = textBoxNombre.Text;
    labelSaludo.Text = "Hola, " + nombre + "!";
}
```

---

6. Eventos y Delegados

Un **evento** permite reaccionar a acciones del usuario (clics, teclas, etc.).

Ejemplo de suscripción manual a un evento:

```
buttonSaludar.Click += new EventHandler(MiMetodo);
private void MiMetodo(object sender, EventArgs e)
{
```

```
    MessageBox.Show("¡Has hecho clic!");
}
```

---

7. Controles comunes

Control	Descripción
Label	Muestra texto estático.
TextBox	Permite ingresar texto.
Button	Ejecuta una acción al hacer clic.
CheckBox	Opción que puede estar activada o no.
RadioButton	Selección única dentro de un grupo.
ComboBox	Lista desplegable con opciones.
ListBox	Lista de elementos seleccionables.
PictureBox	Muestra una imagen.
Timer	Ejecuta eventos cada cierto intervalo.

---

8. Manejo de formularios múltiples

Puedes tener más de un formulario en una aplicación.

Ejemplo: Abrir otro formulario

1. Crear un nuevo formulario: **Agregar → Formulario de Windows Forms → Form2.cs**
2. Código para abrirlo desde `Form1`:

```
Form2 nuevoForm = new Form2();
nuevoForm.Show(); // ShowDialog() para abrir como modal
```

---

9. Buenas prácticas

- Usa nombres descriptivos para controles (`btnGuardar`, `txtNombre`, etc.).
- Divide la lógica en métodos para evitar código duplicado.
- Usa formularios modales cuando necesites confirmar algo antes de continuar.
- Valida siempre los datos introducidos por el usuario.

---

10. Proyecto de práctica

Enunciado: Calculadora de IMC (Índice de Masa Corporal)

Controles necesarios:

- 2 TextBox: Peso (kg) y Altura (m).
- 1 Button: Calcular.
- 1 Label: Resultado.

Lógica:

```
private void btnCalcular_Click(object sender, EventArgs e)
{
    double peso = double.Parse(txtPeso.Text);
    double altura = double.Parse(txtAltura.Text);

    double imc = peso / (altura * altura);
    lblResultado.Text = "Tu IMC es: " + imc.ToString("0.00");

    if (imc < 18.5)
        lblResultado.Text += " (Bajo peso)";
    else if (imc < 25)
        lblResultado.Text += " (Normal)";
    else if (imc < 30)
        lblResultado.Text += " (Sobrepeso)";
    else
        lblResultado.Text += " (Obesidad)";
}
```

---

11. Recursos útiles

- Documentación oficial de Microsoft - Windows Forms
- Curso gratuito en YouTube: "Windows Forms C# desde cero".
- Foros: Stack Overflow, Microsoft Learn.

---

12. Ejercicio final sugerido

Crear una agenda de contactos simple, con los siguientes elementos:

- TextBox para nombre, teléfono y correo.
- Botón para "Agregar contacto".
- ListBox para mostrar los contactos añadidos.
- Botón para "Eliminar contacto seleccionado".

Lección 19: Controles Avanzados en C# con Windows Forms

Objetivos

- Conocer y dominar los **Controles Avanzados** disponibles en Windows Forms.
- Aprender a integrar y manipular controles como `TreeView`, `ListView`, `TabControl`, `DataGridView`, `ProgressBar`, `Timer`, `NotifyIcon`, y `TrackBar`.
- Implementar proyectos reales donde se combinen varios de estos controles.
- Evaluar tus conocimientos con ejercicios y un mini test.

---

1. Introducción a los Controles Avanzados

Los **Controles Avanzados** de Windows Forms permiten mejorar la interactividad, usabilidad y funcionalidad de las interfaces gráficas en aplicaciones de escritorio con C#. Van más allá de los controles básicos como `Button`, `TextBox`, y `Label`.

Algunos ejemplos clave:

Control	Uso Principal
`TreeView`	Visualizar jerarquías (directorios, estructuras, etc.)
`ListView`	Mostrar listas de ítems con detalles y columnas
`DataGridView`	Mostrar y editar datos tabulares
`TabControl`	Agrupar contenido en pestañas
`ProgressBar`	Indicar progreso de tareas
`Timer`	Ejecutar acciones periódicamente
`TrackBar`	Controlar valores numéricos deslizando una barra
`NotifyIcon`	Mostrar icono en la bandeja del sistema

---

2. Control `TreeView`

Uso

Permite mostrar datos jerárquicos (por ejemplo, un sistema de archivos).

Ejemplo

```
TreeNode root = new TreeNode("Animales");
root.Nodes.Add("Mamíferos");
root.Nodes.Add("Aves");
treeView1.Nodes.Add(root);
treeView1.ExpandAll();
```

Propiedades clave

- `Nodes`: colección de nodos del árbol
- `SelectedNode`: nodo seleccionado
- `ExpandAll()`, `CollapseAll()`: expandir o colapsar todo

---

3. Control `ListView`

Uso

Ideal para mostrar listas con múltiples columnas y detalles visuales.

Ejemplo

```
listView1.View = View.Details;
listView1.Columns.Add("Nombre", 100);
listView1.Columns.Add("Edad", 50);
ListViewItem item = new ListViewItem("Laura");
item.SubItems.Add("30");
listView1.Items.Add(item);
```

Modos de vista

- `Details` (columnas)
- `LargeIcon`, `SmallIcon`, `List`, `Tile`

---

4. Control `TabControl`

Uso

Permite organizar contenido en pestañas.

Ejemplo

```
tabControl1.TabPages.Add("Configuración");
tabControl1.TabPages[0].Controls.Add(new Button() { Text = "Guardar" });
```

Propiedades clave

- `TabPages`: colección de pestañas
- `SelectedIndex`: índice de pestaña activa

---

5. Control `DataGridView`

Uso

Mostrar, editar y gestionar tablas de datos.

Ejemplo simple

```
dataGridView1.ColumnCount = 2;
dataGridView1.Columns[0].Name = "Producto";
dataGridView1.Columns[1].Name = "Precio";
dataGridView1.Rows.Add("Pan", "1.20");
```

Funcionalidades

- Edición de celdas
- Vinculación a `DataTable` o `List<>`
- Eventos: `CellClick`, `RowHeaderMouseClick`

---

6. Control `ProgressBar` y `Timer`

Uso

`ProgressBar` muestra progreso visual y `Timer` ejecuta tareas periódicas.

Ejemplo conjunto

```
progressBar1.Maximum = 100;
timer1.Interval = 100;
timer1.Tick += (s, e) => {
    if (progressBar1.Value < 100)
        progressBar1.Value += 5;
    else
        timer1.Stop();
};
timer1.Start();
```

---

7. Control `TrackBar`

Uso

Permite seleccionar valores numéricos arrastrando.

Ejemplo

```
trackBar1.Minimum = 0;
```

```
trackBar1.Maximum = 100;
trackBar1.ValueChanged += (s, e) =>
{
    label1.Text = "Valor: " + trackBar1.Value;
};
```

---

8. Control `NotifyIcon`

Uso

Muestra un icono en la bandeja del sistema con mensajes contextuales.

Ejemplo

```
notifyIcon1.Icon = SystemIcons.Information;
notifyIcon1.Visible = true;
notifyIcon1.BalloonTipTitle = "Atención";
notifyIcon1.BalloonTipText = "Tu aplicación sigue activa";
notifyIcon1.ShowBalloonTip(3000);
```

---

9. Proyecto Integrador: Mini Administrador de Tareas

Funcionalidad

- Mostrar tareas en `DataGridView`
- Añadir tareas con botón
- Ver historial en `ListView`
- Indicador de progreso con `ProgressBar`

Estructura básica

```
// Al cargar el formulario
dataGridView1.Columns.Add("Tarea", "Tarea");
dataGridView1.Columns.Add("Estado", "Estado");

// Botón para agregar tarea
private void btnAgregar_Click(object sender, EventArgs e)
{
    dataGridView1.Rows.Add(txtTarea.Text, "Pendiente");
}
```

Ejercicios Propuestos

1. **Crear un árbol con nodos de Continentes → Países → Ciudades.**
2. **Mostrar una lista de productos con `ListView` en modo `Details`.**
3. **Construir una interfaz con pestañas (una con configuración, otra con estadísticas).**
4. **Simular una barra de carga con `ProgressBar` y `Timer`.**
5. **Usar `TrackBar` para ajustar el volumen de un reproductor simulado.**
6. **Implementar `NotifyIcon` que avise cada vez que se guarda un archivo.**

---

11. Test Final (5 preguntas)

1. ¿Qué control permite mostrar datos en forma tabular con edición de celdas?
a) TreeView
b) ListBox
c) DataGridView
d) TabControl
Respuesta: c)

2. ¿Cuál propiedad de `TreeView` se usa para añadir nodos?
a) Items
b) Controls
c) Nodes
d) List
Respuesta: c)

3. ¿Qué hace el evento `Tick` del `Timer`?
a) Reinicia el formulario
b) Se ejecuta una vez
c) Se ejecuta en intervalos definidos
d) Cambia el color del control
Respuesta: c)

4. ¿Qué control permite cambiar pestañas con contenido?
a) TabControl
b) GroupBox
c) MenuStrip
d) ToolStrip
Respuesta: a)

5. ¿Para qué sirve el `NotifyIcon`?
a) Para mostrar un diálogo
b) Para abrir otro formulario
c) Para mostrar un icono en la barra de título
d) Para mostrar un icono en la bandeja del sistema
Respuesta: d)

Conclusión

Dominar los **Controles Avanzados en C#** permite crear aplicaciones ricas, organizadas y mucho más funcionales. Su correcta combinación facilita un flujo intuitivo para el usuario.

Lección 20: Formularios MDI y Personalización en C#

Objetivo de la lección

- Entender qué son los formularios MDI (Multiple Document Interface).
- Crear una aplicación MDI en C# con Windows Forms.
- Aprender a personalizar el formulario padre e hijo.
- Implementar menús y barras de herramientas en una interfaz MDI.
- Usar estilos visuales y personalización de apariencia.

---

1. ¿Qué es un Formulario MDI?

Un formulario **MDI** permite tener varios formularios hijos abiertos dentro de un mismo contenedor padre. Es útil para aplicaciones como editores de texto, IDEs, o cualquier software que gestione múltiples documentos o vistas simultáneamente.

- MDI: *Multiple Document Interface* (Interfaz de Documentos Múltiples).
- SDI: *Single Document Interface* (Interfaz de Documento Único).

---

2. Crear un Formulario MDI

Paso 1: Crear el formulario principal (MDI Parent)

1. Abre Visual Studio.
2. Crea un nuevo proyecto de tipo **Windows Forms App (.NET Framework)**.
3. Nombra el proyecto `MDIApp`.

Paso 2: Configurar el formulario como MDI

En el formulario principal (Form1), ve a las propiedades y configura:

```
this.IsMdiContainer = true;
this.Text = "Formulario MDI Principal";
this.WindowState = FormWindowState.Maximized;
```

---

3. Crear Formularios Hijos

Agrega un nuevo formulario al proyecto:

1. Clic derecho en el proyecto > Agregar > Windows Form.

2. Nómbralo `FormHijo1.cs`.

Agrega controles si lo deseas (Label, TextBox, etc.).

---

4. Abrir Formularios Hijos desde el Padre

Agrega un **menú** o botón en el formulario principal para abrir formularios hijos:

Agregar un menú Strip:

Arrastra un **MenuStrip** al formulario padre y agrega lo siguiente:

```
Archivo
 └─ Nuevo Formulario
```

Haz doble clic en `Nuevo Formulario` e implementa esto:

```
private void nuevoFormularioToolStripMenuItem_Click(object sender, EventArgs e)
{
    FormHijo1 hijo = new FormHijo1();
    hijo.MdiParent = this;
    hijo.Show();
}
```

---

5. Personalización de los Formularios

Personalizar el formulario padre:

```
this.BackColor = Color.LightSteelBlue;
this.FormBorderStyle = FormBorderStyle.Sizable;
```

Personalizar el formulario hijo:

```
this.Text = "Formulario Hijo";
this.BackColor = Color.Beige;
this.FormBorderStyle = FormBorderStyle.FixedDialog;
this.StartPosition = FormStartPosition.CenterScreen;
```

---

6. Agregar Barras de Herramientas (ToolStrip)

Agrega un **ToolStrip** desde la caja de herramientas al formulario principal.

Agrega botones con íconos y nombres, como:

- Nuevo
- Guardar
- Cerrar

Manejo de evento:

```
private void tsbNuevo_Click(object sender, EventArgs e)
{
    FormHijo1 hijo = new FormHijo1();
    hijo.MdiParent = this;
    hijo.Show();
}
```

---

7. Organización de Formularios Hijos

Puedes organizar los formularios hijos desde el formulario principal:

```
private void cascadaToolStripMenuItem_Click(object sender, EventArgs e)
{
    this.LayoutMdi(MdiLayout.Cascade);
}

private void mosaicoHorizontalToolStripMenuItem_Click(object sender, EventArgs
e)
{
    this.LayoutMdi(MdiLayout.TileHorizontal);
}

private void mosaicoVerticalToolStripMenuItem_Click(object sender, EventArgs e)
{
    this.LayoutMdi(MdiLayout.TileVertical);
}
```

---

8. Personalización Avanzada: Temas y Estilo Visual

Puedes personalizar más usando propiedades o librerías externas como **MetroFramework**, pero en WinForms puro puedes hacer lo siguiente:

Cambiar estilo de controles:

```
button1.FlatStyle = FlatStyle.Flat;
button1.BackColor = Color.CadetBlue;
button1.ForeColor = Color.White;
```

Estilo de fuente:

```
label1.Font = new Font("Segoe UI", 12, FontStyle.Bold);
```

Agregar íconos personalizados:

```
this.Icon = new Icon("icono.ico");
```

 9. Cerrar Todos los Formularios Hijos

Función para cerrar todos los hijos abiertos:

```
private void cerrarTodoToolStripMenuItem_Click(object sender, EventArgs e)
{
    foreach (Form frm in this.MdiChildren)
    {
        frm.Close();
    }
}
```

---

10. Ejercicio Propuesto

Objetivo: Crear una aplicación MDI que simule un editor de documentos con múltiples ventanas.

Requisitos:

- Menú con opciones: Nuevo, Organizar, Cerrar Todo, Salir.
- ToolStrip con botones: Nuevo, Guardar (simulado), Ayuda.
- Formularios hijos que tengan un área de texto (TextBox multiline).
- Personalización visual de formularios hijos (colores, bordes, títulos).
- Organización de ventanas en cascada o mosaico.

Evaluación

Contesta las siguientes preguntas:

1. ¿Qué propiedad convierte un formulario en contenedor MDI?
2. ¿Cómo se asigna un formulario hijo a su padre MDI?
3. ¿Qué métodos se usan para organizar formularios hijos?
4. ¿Qué control se usa para crear menús?
5. ¿Cómo puedes cambiar el fondo de un formulario en WinForms?
6.

Resumen

Tema	Detalles clave
`IsMdiContainer`	Activa el modo MDI en el formulario padre
`MdiParent`	Asigna el padre a un formulario hijo
`LayoutMdi()`	Organiza los formularios hijos
`MenuStrip` y `ToolStrip`	Menús y herramientas
Personalización visual	Colores, fuentes, bordes, íconos

Consejo Final

Los formularios MDI permiten crear interfaces potentes para trabajar con múltiples documentos o vistas. Aunque hoy en día los entornos con pestañas (como en navegadores) son más comunes, **el concepto MDI sigue vigente en herramientas profesionales** y te ayuda a entender mejor la arquitectura de interfaces complejas.

Lección 21: Conexión con Bases de Datos (Windows Forms en C#)

Objetivos de Aprendizaje

- Comprender qué es ADO.NET y cómo se usa en C#.
- Conectar una aplicación de Windows Forms con una base de datos (SQL Server).
- Ejecutar operaciones básicas: insertar, actualizar, eliminar y mostrar datos.
- Usar controles como `DataGridView`, `TextBox`, `Button` y `ComboBox` para manejar datos.

Parte 1: Introducción Teórica

¿Qué es ADO.NET?

ADO.NET es una tecnología de acceso a datos que permite a las aplicaciones .NET comunicarse con bases de datos. Ofrece objetos como:

- `SqlConnection`: para abrir una conexión con una base de datos SQL Server.
- `SqlCommand`: para ejecutar comandos SQL.
- `SqlDataReader`: para leer datos de forma rápida y hacia adelante.
- `SqlDataAdapter`: para llenar tablas en memoria (`DataTable`, `DataSet`).
- `DataTable`: estructura en memoria que representa una tabla.

Requisitos

- Visual Studio (2022 o superior recomendado).
- SQL Server o LocalDB.
- Base de datos de ejemplo: `ContactosDB` con tabla `Contactos`.

Parte 2: Preparar la Base de Datos

Crear la base de datos

```
CREATE DATABASE ContactosDB;


USE ContactosDB;


CREATE TABLE Contactos (
```

```
    Id INT PRIMARY KEY IDENTITY(1,1),
    Nombre NVARCHAR(100),
    Telefono NVARCHAR(20),
    Correo NVARCHAR(100)
);
```

Parte 3: Crear el Proyecto Windows Forms

1. Abre Visual Studio.
2. Crea un nuevo proyecto: `Windows Forms App (.NET Framework)` o `.NET Core` si lo prefieres.
3. Nómbralo: `AppContactos`.

Parte 4: Interfaz de Usuario

Agrega los siguientes controles desde la Caja de Herramientas:

- 3 `TextBox` para `Nombre`, `Teléfono`, `Correo`.
- 4 `Button`: Agregar, Editar, Eliminar, Cargar.
- 1 `DataGridView` para mostrar los contactos.

Diseño recomendado:

Control	Nombre en Código
TextBox Nombre	`txtNombre`
TextBox Teléfono	`txtTelefono`
TextBox Correo	`txtCorreo`
Button Agregar	`btnAgregar`
Button Editar	`btnEditar`
Button Eliminar	`btnEliminar`
Button Cargar	`btnCargar`
DataGridView	`dgvContactos`

Parte 5: Cadena de Conexión

```
string conexion = "Server=.;Database=ContactosDB;Trusted_Connection=True;";
```

Parte 6: Código Paso a Paso

1. Mostrar los datos

```
private void CargarDatos()
{
```

```
    using (SqlConnection con = new SqlConnection(conexion))
    {
        string query = "SELECT * FROM Contactos";
        SqlDataAdapter da = new SqlDataAdapter(query, con);
        DataTable dt = new DataTable();
        da.Fill(dt);
        dgvContactos.DataSource = dt;
    }
}
```

2. Agregar contacto

```
private void btnAgregar_Click(object sender, EventArgs e)
{
    using (SqlConnection con = new SqlConnection(conexion))
    {
        con.Open();
        string query = "INSERT INTO Contactos (Nombre, Telefono, Correo) VALUES 
(@nombre, @telefono, @correo)";
        using (SqlCommand cmd = new SqlCommand(query, con))
        {
            cmd.Parameters.AddWithValue("@nombre", txtNombre.Text);
            cmd.Parameters.AddWithValue("@telefono", txtTelefono.Text);
            cmd.Parameters.AddWithValue("@correo", txtCorreo.Text);
            cmd.ExecuteNonQuery();
        }
        MessageBox.Show("Contacto agregado.");
        CargarDatos();
    }
}
```

3. Seleccionar contacto en el DataGridView

```
private void dgvContactos_CellClick(object sender, DataGridViewCellEventArgs e)
{
    if (e.RowIndex >= 0)
    {
        DataGridViewRow fila = dgvContactos.Rows[e.RowIndex];
        txtNombre.Text = fila.Cells["Nombre"].Value.ToString();
        txtTelefono.Text = fila.Cells["Telefono"].Value.ToString();
```

```
        txtCorreo.Text = fila.Cells["Correo"].Value.ToString();
        contactoId = Convert.ToInt32(fila.Cells["Id"].Value); // Guardar ID
    }
}
int contactoId = 0;
```

4. Editar contacto

```
private void btnEditar_Click(object sender, EventArgs e)
{
    using (SqlConnection con = new SqlConnection(conexion))
    {
        con.Open();
        string query = "UPDATE Contactos SET Nombre=@nombre, Telefono=@telefono,
Correo=@correo WHERE Id=@id";
        using (SqlCommand cmd = new SqlCommand(query, con))
        {
            cmd.Parameters.AddWithValue("@nombre", txtNombre.Text);
            cmd.Parameters.AddWithValue("@telefono", txtTelefono.Text);
            cmd.Parameters.AddWithValue("@correo", txtCorreo.Text);
            cmd.Parameters.AddWithValue("@id", contactoId);
            cmd.ExecuteNonQuery();
        }
        MessageBox.Show("Contacto actualizado.");
        CargarDatos();
    }
}
```

5. Eliminar contacto

```
private void btnEliminar_Click(object sender, EventArgs e)
{
    using (SqlConnection con = new SqlConnection(conexion))
    {
        con.Open();
        string query = "DELETE FROM Contactos WHERE Id=@id";
        using (SqlCommand cmd = new SqlCommand(query, con))
        {
            cmd.Parameters.AddWithValue("@id", contactoId);
            cmd.ExecuteNonQuery();
```

```
        }
        MessageBox.Show("Contacto eliminado.");
        CargarDatos();
    }
}
```

6. Cargar contactos en el botón

```
private void btnCargar_Click(object sender, EventArgs e)
{
    CargarDatos();
}
```

Parte 7: Mejores Prácticas

- Usa `using` para liberar recursos automáticamente.
- Valida que los campos no estén vacíos antes de insertar o actualizar.
- Usa `try-catch` para manejar errores de conexión o SQL.
- Usa capas separadas (DAO, DTO) en proyectos grandes para mejor mantenimiento.

Parte 8: Ejercicios Propuestos

1. Añade un campo "Dirección" en la tabla y en el formulario.
2. Ordena los resultados alfabéticamente por nombre.
3. Implementa búsqueda por nombre con `LIKE`.
4. Muestra la cantidad total de contactos en una `Label`.
5. Usa `ComboBox` para filtrar contactos por dominio de correo (`@gmail.com`, etc.).

Conclusión

Conectar Windows Forms a una base de datos te permite crear aplicaciones reales con persistencia de datos. Lo aprendido aquí es la base de muchas aplicaciones empresariales. Puedes escalar este proyecto fácilmente hacia capas más robustas, agregar autenticación, o migrar a Entity Framework.

Parte V – Desarrollo Web con ASP.NET

Lección 22: Introducción a ASP.NET con C#

Objetivos de la lección

Al finalizar esta lección, el estudiante será capaz de:

- Comprender qué es ASP.NET y su relación con C#.
- Identificar las diferentes tecnologías dentro de ASP.NET.
- Crear su primera aplicación web con ASP.NET Core.
- Entender el ciclo de vida de una aplicación ASP.NET.
- Utilizar controladores, vistas y modelos (patrón MVC básico).
- Realizar peticiones HTTP simples y mostrar datos en una vista.

---

1. ¿Qué es ASP.NET?

ASP.NET es un marco (framework) de desarrollo web creado por Microsoft para construir aplicaciones web modernas y servicios web. Se ejecuta sobre el .NET Framework (ASP.NET clásico) o sobre .NET Core (ASP.NET Core).

> **ASP.NET Core** es la evolución moderna, multiplataforma (Windows, Linux y macOS), modular y más rápida.

Características principales

- Soporte para **MVC (Model-View-Controller)**.
- Enrutamiento URL.
- Seguridad integrada (autenticación/autorización).
- Soporte para APIs RESTful.
- Integración con Entity Framework para bases de datos.
- Inyección de dependencias incorporada.
- Escalable y de alto rendimiento.

---

2. Entorno de desarrollo

Requisitos

- Visual Studio 2022 o superior (recomendado) o Visual Studio Code.
- .NET SDK (.NET 6 o .NET 8).

- Navegador web moderno (Chrome, Edge, Firefox, etc.).

Comandos para instalar el SDK y crear proyecto:

```
dotnet new --install Microsoft.AspNetCore.App
dotnet new mvc -n MiPrimeraAppWeb
cd MiPrimeraAppWeb
dotnet run
```

---

3. Estructura de un proyecto ASP.NET Core MVC

```
MiPrimeraAppWeb/
├── Controllers/      <- Controladores
├── Models/           <- Modelos
├── Views/            <- Vistas (Razor)
├── wwwroot/          <- Archivos estáticos (JS, CSS, imágenes)
├── Program.cs        <- Punto de entrada
├── appsettings.json  <- Configuración
```

---

4. Patrón MVC explicado

Modelo (Model)

Contiene la lógica de negocio y los datos.

```
public class Producto
{
    public int Id { get; set; }
    public string Nombre { get; set; }
    public decimal Precio { get; set; }
}
```

Vista (View)

Interfaz que muestra la información al usuario, escrita con Razor (`.cshtml`).

```
@model Producto
<h2>@Model.Nombre</h2>
<p>Precio: @Model.Precio €</p>
```

Controlador (Controller)

Lógica que responde a las solicitudes del navegador.

```
public class ProductoController : Controller
{
```

```
    public IActionResult Detalle()
    {
        var producto = new Producto { Id = 1, Nombre = "Teclado", Precio = 35.5M
};
        return View(producto);
    }
}
```

---

5. El ciclo de vida de una solicitud HTTP

1. El navegador envía una solicitud (por ejemplo, `/Producto/Detalle`).
2. El middleware enruta la solicitud al controlador correcto.
3. El controlador procesa y devuelve un modelo a una vista.
4. La vista genera HTML que se devuelve al navegador.

---

6. Primer ejemplo completo

Paso 1: Crear modelo

Archivo: Models/Producto.cs

```
namespace MiPrimeraAppWeb.Models
{
    public class Producto
    {
        public int Id { get; set; }
        public string Nombre { get; set; }
        public decimal Precio { get; set; }
    }
}
```

Paso 2: Crear controlador

Archivo: Controllers/ProductoController.cs

```
using Microsoft.AspNetCore.Mvc;
using MiPrimeraAppWeb.Models;

namespace MiPrimeraAppWeb.Controllers
{
    public class ProductoController : Controller
    {
```

```
        public IActionResult Detalle()
        {
            var producto = new Producto
            {
                Id = 1,
                Nombre = "Teclado mecánico",
                Precio = 45.90M
            };
            return View(producto);
        }
    }
}
```

Paso 3: Crear vista

Archivo: Views/Producto/Detalle.cshtml

```
@model MiPrimeraAppWeb.Models.Producto

<h1>Detalles del Producto</h1>
<p><strong>Nombre:</strong> @Model.Nombre</p>
<p><strong>Precio:</strong> @Model.Precio €</p>
```

Paso 4: Ejecutar y visitar

Ejecuta `dotnet run` y visita:
`http://localhost:5000/Producto/Detalle`

---

7. Introducción a la seguridad

ASP.NET ofrece mecanismos de autenticación como:

- Cookie Authentication
- JWT (JSON Web Tokens)
- Identity (sistema completo de gestión de usuarios)

  ⚠ En esta lección no se profundiza, pero se puede activar fácilmente con `Individual User Accounts` al crear el proyecto.

---

8. Buenas prácticas iniciales

- Separar responsabilidades correctamente (MVC).
- Usar servicios en controladores (inyección de dependencias).

- Validar los datos en los modelos.
- Manejar errores con páginas personalizadas.

---

9. Ejercicios

Ejercicio 1:

Crea un modelo llamado `Libro` con propiedades: `Id`, `Titulo`, `Autor`, `Precio`.

Ejercicio 2:

Crea un controlador `LibroController` con una acción `Vista` que devuelva un objeto `Libro` a la vista.

Ejercicio 3:

Diseña una vista Razor `Vista.cshtml` que muestre los datos del libro con formato HTML.

---

10. Recursos adicionales

- Documentación oficial ASP.NET Core
- Curso gratuito de Microsoft Learn: ASP.NET Core fundamentals
- GitHub: ejemplos y plantillas ASP.NET

Conclusión

ASP.NET es una poderosa plataforma para desarrollar aplicaciones web modernas. Esta introducción te brinda los conocimientos esenciales para comenzar con proyectos web usando C# y el patrón MVC. En lecciones posteriores podrás explorar bases de datos, APIs REST, autenticación y despliegue.

Lección 23: Introducción a ASP.NET MVC con C#

Objetivos de la Lección

Al finalizar esta lección, podrás:

- Comprender qué es ASP.NET MVC y su arquitectura.
- Crear un proyecto ASP.NET MVC en Visual Studio.
- Entender el funcionamiento del patrón Modelo-Vista-Controlador.
- Usar rutas, controladores, vistas y modelos.
- Conectar con bases de datos mediante Entity Framework.
- Implementar operaciones CRUD.
- Aplicar validaciones y usar formularios.
- Publicar una aplicación ASP.NET MVC.

---

1. ¿Qué es ASP.NET MVC?

Definición

ASP.NET MVC (Model-View-Controller) es un framework de desarrollo web que separa la aplicación en tres componentes principales: **Modelo, Vista y Controlador**, lo cual mejora la organización del código y facilita el mantenimiento y escalabilidad.

Características principales

- Separación de responsabilidades (SoC).
- URLs amigables y enrutamiento personalizado.
- Soporte completo para HTML, CSS, JavaScript, AJAX.
- Integración con Entity Framework.
- Testing más sencillo (gracias a su arquitectura desacoplada).

---

2. Arquitectura MVC

Modelo

Representa los datos y la lógica de negocio. Puede incluir clases que interactúan con bases de datos.

Vista

Es la interfaz de usuario, representada por archivos `.cshtml` que combinan HTML y Razor (sintaxis C# para vistas).

Controlador

Maneja las peticiones del usuario, interactúa con el modelo y selecciona la vista adecuada.

---

3. Crear un Proyecto ASP.NET MVC

Requisitos Previos

- Visual Studio 2019 o superior
- .NET Framework o .NET Core SDK
- SQL Server Express (opcional)

Pasos para crear el proyecto

1. Abre Visual Studio → `Crear nuevo proyecto`.
2. Selecciona "ASP.NET Web Application (.NET Framework)" o "ASP.NET Core Web App (Model-View-Controller)".
3. Nómbralo `MiPrimeraAppMVC`.
4. Elige "MVC" como plantilla del proyecto.
5. Haz clic en Crear.

---

4. Estructura del Proyecto

```
MiPrimeraAppMVC/
|
├── Controllers/       → Controladores (lógica)
├── Models/            → Modelos (datos y lógica de negocio)
├── Views/             → Vistas (interfaz HTML)
├── Views/Shared/      → Vistas comunes (layout, error)
├── App_Start/         → Configuraciones (RouteConfig)
├── Web.config         → Configuración del sitio
├── Global.asax        → Configuración global
```

---

5. Enrutamiento en MVC

Ubicado en `App_Start/RouteConfig.cs`:

```
routes.MapRoute(
```

```
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional }
);
```

- URL `/Home/Index` llama a `HomeController` y su método `Index()`.

---

6. Creación de un Controlador

Código de ejemplo: `Controllers/ProductoController.cs`

```
using System.Web.Mvc;


namespace MiPrimeraAppMVC.Controllers
{
    public class ProductoController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Esto creará la vista correspondiente en `Views/Producto/Index.cshtml`.

---

7. Crear un Modelo

Código de ejemplo: `Models/Producto.cs`

```
namespace MiPrimeraAppMVC.Models
{
    public class Producto
    {
        public int Id { get; set; }
        public string Nombre { get; set; }
        public decimal Precio { get; set; }
    }
}
```

---

8. Crear una Vista

Razor View (`Views/Producto/Index.cshtml`)

```
@model List<MiPrimeraAppMVC.Models.Producto>


<h2>Lista de productos</h2>


<ul>
@foreach (var p in Model)
{
    <li>@p.Nombre - $@p.Precio</li>
}
</ul>
```

---

9. Conexión a Base de Datos con Entity Framework

Instalar paquetes NuGet

```
Install-Package EntityFramework
```

Crear DbContext

```
using System.Data.Entity;


public class AppDbContext : DbContext
{
    public DbSet<Producto> Productos { get; set; }
}
```

Agregar conexión en `Web.config`

```
<connectionStrings>
  <add name="AppDbContext" connectionString="Data Source=.;Initial Catalog=MiAppDB;Integrated Security=True" providerName="System.Data.SqlClient" />
</connectionStrings>
```

---

10. Operaciones CRUD

Crear (Create)

```
[HttpPost]
public ActionResult Create(Producto producto)
```

```
{
    using (var db = new AppDbContext())
    {
        db.Productos.Add(producto);
        db.SaveChanges();
    }
    return RedirectToAction("Index");
}
```

📄 Vista de Create (`Create.cshtml`)

```
@model MiPrimeraAppMVC.Models.Producto

@using (Html.BeginForm())
{
    <label>Nombre:</label> @Html.TextBoxFor(p => p.Nombre) <br />
    <label>Precio:</label> @Html.TextBoxFor(p => p.Precio) <br />
    <input type="submit" value="Guardar" />
}
```

Leer (Index)

```
public ActionResult Index()
{
    using (var db = new AppDbContext())
    {
        var productos = db.Productos.ToList();
        return View(productos);
    }
}
```

Editar (Edit)

```
public ActionResult Edit(int id)
{
    using (var db = new AppDbContext())
    {
        var prod = db.Productos.Find(id);
        return View(prod);
    }
}
```

```
[HttpPost]
public ActionResult Edit(Producto p)
{
    using (var db = new AppDbContext())
    {
        db.Entry(p).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
}
```

Eliminar (Delete)

```
public ActionResult Delete(int id)
{
    using (var db = new AppDbContext())
    {
        var prod = db.Productos.Find(id);
        db.Productos.Remove(prod);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
}
```

---

11. Validación de Formularios

Atributos en el modelo

```
using System.ComponentModel.DataAnnotations;

public class Producto
{
    public int Id { get; set; }

    [Required]
    public string Nombre { get; set; }

    [Range(0.01, 9999.99)]
```

```
    public decimal Precio { get; set; }
}
```

Mostrar validaciones en la vista

```
@Html.ValidationSummary()
@Html.TextBoxFor(p => p.Nombre)
@Html.ValidationMessageFor(p => p.Nombre)
```

---

12. Publicar la Aplicación

Opciones de publicación

- IIS local
- Azure App Services
- FTP / Hosting externo

Pasos básicos

1. Clic derecho en proyecto → `Publicar`.
2. Selecciona método (IIS, Azure, carpeta, etc.).
3. Configura conexión y destino.
4. Publicar.

Recursos adicionales

- Documentación oficial: https://learn.microsoft.com/en-us/aspnet/mvc
- Tutoriales en Microsoft Learn.
- Libros recomendados: *Pro ASP.NET MVC 5*, *ASP.NET Core in Action*.

Ejercicio Final Propuesto

Crea una aplicación MVC llamada `LibreriaWeb`, que tenga:

- Un modelo `Libro` con propiedades: `Id`, `Titulo`, `Autor`, `Año`, `Precio`.
- Controlador con acciones: `Index`, `Create`, `Edit`, `Delete`, `Details`.
- Vistas completas con validaciones.
- Conexión a base de datos con Entity Framework.

Lección 24: Introducción y Desarrollo con ASP.NET Core

Objetivos de la Lección

- Comprender qué es ASP.NET Core y cómo se diferencia del ASP.NET tradicional.
- Crear un proyecto básico con ASP.NET Core.
- Comprender la estructura del proyecto.
- Manejar rutas y controladores (MVC).
- Usar Razor Pages.
- Realizar operaciones CRUD básicas.
- Conectar con una base de datos mediante Entity Framework Core.
- Implementar inyección de dependencias.
- Hacer pruebas básicas con Postman.

---

1. ¿Qué es ASP.NET Core?

ASP.NET Core es un **framework web open source, multiplataforma y modular** desarrollado por Microsoft. Permite construir **aplicaciones web modernas, APIs REST, microservicios y más**, ejecutables en Windows, macOS y Linux.

✳ Características Clave

- Multiplataforma (Windows, Linux, macOS)
- Basado en .NET Core
- Alto rendimiento
- Inyección de dependencias integrada
- Soporte para middleware
- Compatibilidad con Razor Pages, MVC y Web API

---

2. Crear un Proyecto ASP.NET Core

Requisitos Previos

- .NET SDK instalado: https://dotnet.microsoft.com/download

- Visual Studio o VS Code

Comando para crear un proyecto MVC:

```
dotnet new mvc -n MiPrimeraWebApp
cd MiPrimeraWebApp
dotnet run
```

O en Visual Studio:

1. Nuevo proyecto → ASP.NET Core Web App (Model-View-Controller)
2. Nombre: `MiPrimeraWebApp`
3. Asegúrate de seleccionar `.NET 6` o superior

---

3. Estructura del Proyecto

```
MiPrimeraWebApp/
│
├── Controllers/        ← Controladores MVC
├── Models/             ← Clases de dominio / datos
├── Views/              ← Archivos .cshtml (Razor)
├── wwwroot/            ← Archivos estáticos (CSS, JS, etc.)
├── Program.cs          ← Punto de entrada de la app
├── appsettings.json    ← Configuración de la app
```

---

4. Primer Controlador y Vista

Crear un Controlador

```
// Controllers/HomeController.cs
using Microsoft.AspNetCore.Mvc;

public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    public IActionResult AcercaDe()
    {
```

```
        ViewData["Mensaje"] = "Esta es una app de prueba.";
        return View();
    }
}
```

Crear una Vista

```
<!-- Views/Home/Index.cshtml -->
@{
    ViewData["Title"] = "Inicio";
}
<h1>Bienvenido a MiPrimeraWebApp</h1>
```

---

5. Enrutamiento (Routing)

ASP.NET Core utiliza **routing basado en atributos** y **routing convencional**.

En `Program.cs`:

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

---

6. Uso de Razor Pages (Alternativa a MVC)

```
dotnet new razor -n MiRazorApp
cd MiRazorApp
dotnet run
```

Un archivo Razor combina HTML y C#:

```
<!-- Pages/Index.cshtml -->
@page
@model IndexModel
@{
    ViewData["Title"] = "Inicio";
}
<h2>Hola desde Razor Pages</h2>
```

---

7. CRUD Básico con Entity Framework Core

Paso 1: Crear el Modelo

```
// Models/Producto.cs
```

```
public class Producto
{
    public int Id { get; set; }
    public string Nombre { get; set; }
    public decimal Precio { get; set; }
}
```

Paso 2: Crear el Contexto

```
// Data/AppDbContext.cs
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public DbSet<Producto> Productos { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
{ }
}
```

Paso 3: Registrar en `Program.cs`

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseInMemoryDatabase("BaseDeDatos"));
```

Paso 4: Crear Controlador CRUD

```
// Controllers/ProductoController.cs
public class ProductoController : Controller
{
    private readonly AppDbContext _context;

    public ProductoController(AppDbContext context)
    {
        _context = context;
    }

    public IActionResult Index() => View(_context.Productos.ToList());

    public IActionResult Crear() => View();

    [HttpPost]
```

```
    public IActionResult Crear(Producto producto)
    {
        _context.Productos.Add(producto);
        _context.SaveChanges();
        return RedirectToAction("Index");
    }
}
```

Paso 5: Vistas Razor para Crear y Listar

```
<!-- Views/Producto/Index.cshtml -->
@model List<Producto>
<h2>Productos</h2>
<a href="/Producto/Crear">Crear nuevo</a>
<ul>
@foreach (var p in Model)
{
    <li>@p.Nombre - $@p.Precio</li>
}
</ul>
<!-- Views/Producto/Crear.cshtml -->
@model Producto
<form asp-action="Crear" method="post">
    <input asp-for="Nombre" placeholder="Nombre" />
    <input asp-for="Precio" placeholder="Precio" />
    <button type="submit">Guardar</button>
</form>
```

---

8. Inyección de Dependencias

```
// Interfaces/IProductoService.cs
public interface IProductoService
{
    List<Producto> ObtenerTodos();
}

// Servicios/ProductoService.cs
public class ProductoService : IProductoService
{
```

```
    private readonly AppDbContext _context;

    public ProductoService(AppDbContext context)
    {
        _context = context;
    }

    public List<Producto> ObtenerTodos()
    {
        return _context.Productos.ToList();
    }
}
```

En `Program.cs`:

```
builder.Services.AddScoped<IProductoService, ProductoService>();
```

---

9. Consumo de API con Postman

Si quieres que el controlador devuelva JSON:

```
[ApiController]
[Route("api/[controller]")]
public class ProductoApiController : ControllerBase
{
    private readonly AppDbContext _context;

    public ProductoApiController(AppDbContext context)
    {
        _context = context;
    }

    [HttpGet]
    public IActionResult Get() => Ok(_context.Productos.ToList());
}
```

---

10. Ejercicios Propuestos

Ejercicio 1

Crea una app Razor Pages con una lista de tareas (ToDo List), permitiendo añadir y eliminar tareas.

Ejercicio 2

Extiende la app anterior usando Entity Framework Core con una base de datos SQLite.

Ejercicio 3

Crea un Web API que permita consultar libros por título y autor. Usa Postman para probar los endpoints.

Ejercicio 4

Integra Bootstrap 5 para mejorar el diseño de las vistas en una app ASP.NET Core MVC.

Recursos Recomendados

- Documentación oficial de ASP.NET Core
- Curso gratuito de Microsoft Learn
- ASP.NET Core GitHub

Conclusión

ASP.NET Core es una herramienta poderosa para desarrollar aplicaciones web modernas, robustas y seguras. Dominar su arquitectura basada en controladores, Razor Pages, Entity Framework y buenas prácticas como la inyección de dependencias te permitirá construir soluciones profesionales escalables.

Lección 25: Entity Framework Core en C#

Índice

---

1. ¿Qué es Entity Framework Core?

Entity Framework Core es un **ORM (Object-Relational Mapper)** para .NET. Permite interactuar con bases de datos relacionales usando objetos C# en lugar de SQL explícito.

🔹 Características:

- Soporte para múltiples motores: SQL Server, SQLite, PostgreSQL, MySQL, etc.
- Compatible con .NET Core y .NET 5/6/7+
- Migrations para mantener la base de datos sincronizada con el modelo

---

2. Ventajas y desventajas

Ventajas:

- Abstracción de la base de datos
- Reducción de código SQL

- Mantenimiento más sencillo del modelo
- Tipado fuerte

Desventajas:

- Menor control sobre consultas complejas
- Potencial sobrecarga de rendimiento si no se configura correctamente

---

3. Instalación y configuración

Paso 1: Crear un proyecto

```
dotnet new console -n EFDemo
cd EFDemo
```

Paso 2: Instalar paquetes EF Core (ejemplo con SQL Server)

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Design
```

---

4. Primer modelo de datos

Clase `Product`

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

DbContext

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlServer("Server=(localdb)\
\mssqllocaldb;Database=EFDemoDb;Trusted_Connection=True;");
}
```

---

5. Migrations y creación de base de datos

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

Esto genera:

- Archivos de migración en `/Migrations`
- Base de datos real en SQL Server LocalDB

---

6. CRUD completo con EF Core

Create

```
using var db = new AppDbContext();
db.Products.Add(new Product { Name = "Teclado", Price = 49.99m });
db.SaveChanges();
```

Read

```
var products = db.Products.ToList();
foreach (var product in products)
    Console.WriteLine($"{product.Id}: {product.Name} - {product.Price}");
```

Update

```
var prod = db.Products.First();
prod.Price = 59.99m;
db.SaveChanges();
```

Delete

```
var prod = db.Products.First();
db.Products.Remove(prod);
db.SaveChanges();
```

---

7. Relaciones entre entidades

🔹 Uno a muchos

```
public class Category
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Product> Products { get; set; }
}
```

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Category Category { get; set; }
}
```

Y se añade `DbSet<Category>` al `AppDbContext`.

---

8. Consultas avanzadas con LINQ

```
// Filtrar
var baratos = db.Products.Where(p => p.Price < 100).ToList();

// Ordenar
var ordenados = db.Products.OrderBy(p => p.Name).ToList();

// Contar
var total = db.Products.Count();

// Incluir relaciones
var productosConCategoria = db.Products.Include(p => p.Category).ToList();
```

---

9. Lazy, Eager y Explicit Loading

- **Lazy Loading**: Se carga automáticamente cuando se accede (requiere configuración especial).
- **Eager Loading**: Usa `.Include()`.
- **Explicit Loading**: Usa `.Entry().Collection().Load()`.

---

10. Validación y restricciones

Anotaciones

```
using System.ComponentModel.DataAnnotations;

public class Product
{
    public int Id { get; set; }
```

```
    [Required, MaxLength(100)]
    public string Name { get; set; }

    [Range(0, 10000)]
    public decimal Price { get; set; }
}
```

---

11. Manejo de errores y transacciones

Manejo de errores

```
try
{
    db.SaveChanges();
}
catch (DbUpdateException ex)
{
    Console.WriteLine("Error al guardar en la base de datos: " + ex.Message);
}
```

Transacciones

```
using var transaction = db.Database.BeginTransaction();
try
{
    db.Products.Add(new Product { Name = "Monitor", Price = 200 });
    db.SaveChanges();
    transaction.Commit();
}
catch
{
    transaction.Rollback();
}
```

---

12. Buenas prácticas

- Usar clases DTO para entrada/salida
- No exponer directamente entidades
- Usar `using` para el contexto o inyectarlo con DI

- Validar datos antes de persistir
- Revisar rendimiento (consultas innecesarias, carga de relaciones)

---

13. Ejercicios prácticos

1. Crear un sistema de usuarios con roles
2. Agregar validaciones a las entidades
3. Crear una relación muchos a muchos entre cursos y estudiantes
4. Implementar paginación y ordenación con LINQ
5. Realizar migraciones con cambios en el modelo

---

14. Proyecto final sugerido

Proyecto: Sistema de Biblioteca

- **Entidades**: Libro, Autor, Usuario, Préstamo
- **Relaciones**:
    - Un autor tiene muchos libros
    - Un usuario puede tener muchos préstamos
- **Funcionalidades**:
    - Registrar libros
    - Buscar libros por autor
    - Realizar y devolver préstamos
    - Ver historial de préstamos de un usuario

Parte VI – Acceso a Datos y Archivos

Lección 26: Archivos y Streams en C#

Objetivos

- Comprender qué son los Streams en .NET.
- Leer y escribir archivos de texto.
- Trabajar con archivos binarios.
- Utilizar clases del espacio de nombres `System.IO`.
- Aprender buenas prácticas de manejo de archivos.

---

1. Introducción a Archivos y Streams

¿Qué es un Stream?

Un **stream** es una secuencia de bytes que representa datos de entrada o salida. En C#, se usan para:

- Leer datos desde un archivo.
- Escribir datos en un archivo.
- Leer o escribir desde memorias, sockets, etc.

Tipos comunes de Stream:

- `FileStream` – Para archivos.
- `MemoryStream` – Para memoria.
- `NetworkStream` – Para red.

---

2. Espacio de nombres `System.IO`

Clases más usadas:

Clase	Descripción
`File`	Métodos estáticos para manipulación de archivos
`FileInfo`	Métodos orientados a objetos para archivos
`Directory`	Métodos estáticos para directorios
`DirectoryInfo`	Versión orientada a objetos para directorios
`StreamReader`	Lee caracteres de un archivo
`StreamWriter`	Escribe caracteres en un archivo
`BinaryReader` / `BinaryWriter`	Para lectura/escritura binaria

---

3. Leer y Escribir Archivos de Texto

Escribir con `StreamWriter`

```
using System.IO;

string ruta = "datos.txt";
using (StreamWriter writer = new StreamWriter(ruta))
{
    writer.WriteLine("Primera línea");
    writer.WriteLine("Segunda línea");
}
```

Leer con `StreamReader`

```
using System.IO;

string ruta = "datos.txt";
using (StreamReader reader = new StreamReader(ruta))
{
    string linea;
    while ((linea = reader.ReadLine()) != null)
    {
        Console.WriteLine(linea);
    }
}
```

> **Nota**: `using` asegura que el archivo se cierre automáticamente.

---

4. Leer y Escribir Archivos Binarios

Escribir con `BinaryWriter`

```
using (BinaryWriter bw = new BinaryWriter(File.Open("datos.bin", 
FileMode.Create)))
{
    bw.Write(10);        // Entero
    bw.Write(3.14);      // Double
    bw.Write("Texto");   // String
}
```

Leer con `BinaryReader`

```
using (BinaryReader br = new BinaryReader(File.Open("datos.bin", FileMode.Open)))
{
    int n = br.ReadInt32();
    double d = br.ReadDouble();
    string texto = br.ReadString();

    Console.WriteLine($"{n}, {d}, {texto}");
}
```

---

5. Otras clases útiles: `File` y `FileInfo`

Leer y escribir con `File`

```
File.WriteAllText("saludo.txt", "Hola mundo!");
string contenido = File.ReadAllText("saludo.txt");
Console.WriteLine(contenido);
```

Leer todas las líneas

```
string[] lineas = File.ReadAllLines("archivo.txt");
foreach (var linea in lineas)
{
    Console.WriteLine(linea);
}
```

---

6. Comprobar existencia y eliminar archivos

```
string archivo = "ejemplo.txt";

if (File.Exists(archivo))
{
    Console.WriteLine("El archivo existe.");
    File.Delete(archivo);
    Console.WriteLine("Archivo eliminado.");
}
else
{
    Console.WriteLine("El archivo no existe.");
```

```
}
```

---

7. Manejo de Directorios

```
string carpeta = "MiCarpeta";

if (!Directory.Exists(carpeta))
{
    Directory.CreateDirectory(carpeta);
    Console.WriteLine("Carpeta creada.");
}
else
{
    Console.WriteLine("Ya existe.");
}
```

Listar archivos:

```
string[] archivos = Directory.GetFiles("MiCarpeta");
foreach (string ruta in archivos)
{
    Console.WriteLine(Path.GetFileName(ruta));
}
```

---

8. Buenas prácticas

1. **Siempre cerrar el archivo** (`using` lo hace por ti).
2. **Manejar excepciones**:
   ```
   try
   {
       // Código de archivo
   }
   catch (IOException ex)
   {
       Console.WriteLine("Error: " + ex.Message);
   }
   ```
3. **Evitar sobrescribir sin aviso**.
4. **Verificar existencia de archivos/directorios antes de acceder**.

---

9. Ejercicio práctico 1: Registro de notas

Enunciado:

Crea una aplicación que permita guardar y leer las notas de alumnos en un archivo de texto. Cada línea debe contener: `Nombre;Nota`.

Solución:

```
string ruta = "notas.txt";

// Guardar datos
using (StreamWriter sw = new StreamWriter(ruta, true))
{
    sw.WriteLine("Ana;9.2");
    sw.WriteLine("Luis;7.8");
}

// Leer datos
using (StreamReader sr = new StreamReader(ruta))
{
    string linea;
    while ((linea = sr.ReadLine()) != null)
    {
        string[] partes = linea.Split(';');
        Console.WriteLine($"Nombre: {partes[0]}, Nota: {partes[1]}");
    }
}
```

---

10. Ejercicio práctico 2: Almacenamiento binario de usuarios

Enunciado:

Guarda los nombres y edades de 3 personas en un archivo binario. Luego lee los datos y los muestra por pantalla.

Solución:

```
string archivo = "usuarios.bin";

// Escribir
using (BinaryWriter bw = new BinaryWriter(File.Open(archivo, FileMode.Create)))
```

```
{
    bw.Write("María");
    bw.Write(30);
    bw.Write("Juan");
    bw.Write(25);
    bw.Write("Pedro");
    bw.Write(40);
}

// Leer
using (BinaryReader br = new BinaryReader(File.Open(archivo, FileMode.Open)))
{
    for (int i = 0; i < 3; i++)
    {
        string nombre = br.ReadString();
        int edad = br.ReadInt32();
        Console.WriteLine($"{nombre} tiene {edad} años");
    }
}
```

---

11. Resumen

Acción	Método o Clase
Leer texto	`StreamReader`, `File.ReadAllText`
Escribir texto	`StreamWriter`, `File.WriteAllText`
Leer binario	`BinaryReader`
Escribir binario	`BinaryWriter`
Crear archivo	`File.Create`
Verificar existencia	`File.Exists`, `Directory.Exists`
Crear directorio	`Directory.CreateDirectory`

---

12. Tarea final

Crea un programa que:

1. Permita registrar tareas pendientes en un archivo de texto.
2. Cada tarea incluye: nombre, prioridad (1-5), y fecha límite.
3. Al iniciar, el programa muestra todas las tareas almacenadas.

Lección 27: Bases de Datos con ADO.NET en C#

Objetivos de la Lección

- Comprender qué es ADO.NET y su papel en C#.
- Aprender a establecer una conexión con una base de datos (SQL Server).
- Ejecutar consultas (SELECT, INSERT, UPDATE, DELETE).
- Utilizar objetos como `SqlConnection`, `SqlCommand`, `SqlDataReader`, `SqlDataAdapter`, `DataSet` y `DataTable`.
- Aplicar buenas prácticas al trabajar con ADO.NET.

---

1. ¿Qué es ADO.NET?

ADO.NET (ActiveX Data Objects .NET) es una tecnología de acceso a datos de .NET que permite interactuar con diversas fuentes de datos, especialmente bases de datos relacionales como SQL Server, MySQL, Oracle, etc.

ADO.NET proporciona:

- **Conexiones** a bases de datos.
- **Comandos** SQL.
- **Lectura y escritura** de datos.
- **Manejo desconectado** con `DataSet` y `DataTable`.

---

2. Requisitos previos

- Visual Studio instalado.
- SQL Server Express o LocalDB.
- Base de datos de prueba (puedes usar la base de datos `Alumnos` con una tabla `Estudiantes`).

Base de datos de ejemplo

```
CREATE DATABASE Alumnos;
GO

USE Alumnos;
GO
```

```sql
CREATE TABLE Estudiantes (
    Id INT PRIMARY KEY IDENTITY,
    Nombre NVARCHAR(100),
    Edad INT,
    Carrera NVARCHAR(50)
);
```

---

3. Conexión a la base de datos con `SqlConnection`

```csharp
using System;
using System.Data.SqlClient;

class Program {
    static void Main() {
        string cadenaConexion =
"Server=localhost;Database=Alumnos;Trusted_Connection=True;";
        using (SqlConnection conexion = new SqlConnection(cadenaConexion)) {
            try {
                conexion.Open();
                Console.WriteLine("Conexión exitosa.");
            } catch (Exception ex) {
                Console.WriteLine("Error al conectar: " + ex.Message);
            }
        }
    }
}
```

> **Nota:** Usa `using` para asegurar que la conexión se cierre correctamente.

---

4. Insertar datos con `SqlCommand`

```csharp
string sql = "INSERT INTO Estudiantes (Nombre, Edad, Carrera) VALUES (@nombre,
@edad, @carrera)";
using (SqlCommand comando = new SqlCommand(sql, conexion)) {
    comando.Parameters.AddWithValue("@nombre", "Laura");
    comando.Parameters.AddWithValue("@edad", 21);
    comando.Parameters.AddWithValue("@carrera", "Informática");

    int filasAfectadas = comando.ExecuteNonQuery();
```

```
    Console.WriteLine("Filas insertadas: " + filasAfectadas);
}
```

---

5. Leer datos con `SqlDataReader`

```
string sql = "SELECT * FROM Estudiantes";
using (SqlCommand comando = new SqlCommand(sql, conexion)) {
    using (SqlDataReader lector = comando.ExecuteReader()) {
        while (lector.Read()) {
            Console.WriteLine($"ID: {lector["Id"]}, Nombre: {lector["Nombre"]}, Edad: {lector["Edad"]}, Carrera: {lector["Carrera"]}");
        }
    }
}
```

🛠️ Actualizar

```
string sql = "UPDATE Estudiantes SET Edad = @edad WHERE Nombre = @nombre";
using (SqlCommand comando = new SqlCommand(sql, conexion)) {
    comando.Parameters.AddWithValue("@edad", 22);
    comando.Parameters.AddWithValue("@nombre", "Laura");

    int filas = comando.ExecuteNonQuery();
    Console.WriteLine("Filas actualizadas: " + filas);
}
```

Eliminar

```
string sql = "DELETE FROM Estudiantes WHERE Nombre = @nombre";
using (SqlCommand comando = new SqlCommand(sql, conexion)) {
    comando.Parameters.AddWithValue("@nombre", "Laura");

    int filas = comando.ExecuteNonQuery();
    Console.WriteLine("Filas eliminadas: " + filas);
}
```

---

7. Uso de `SqlDataAdapter` y `DataTable`

Permite trabajar en modo **desconectado**.

```
using System.Data;
```

```
SqlDataAdapter adaptador = new SqlDataAdapter("SELECT * FROM Estudiantes", conexion);
DataTable tabla = new DataTable();
adaptador.Fill(tabla);

foreach (DataRow fila in tabla.Rows) {
    Console.WriteLine($"Nombre: {fila["Nombre"]}, Edad: {fila["Edad"]}");
}
```

---

8. `DataSet`: varias tablas

```
DataSet dataset = new DataSet();
SqlDataAdapter adaptador = new SqlDataAdapter("SELECT * FROM Estudiantes", conexion);
adaptador.Fill(dataset, "Estudiantes");

foreach (DataRow fila in dataset.Tables["Estudiantes"].Rows) {
    Console.WriteLine($"Nombre: {fila["Nombre"]}");
}
```

---

9. Buenas prácticas

- Siempre usa `using` para cerrar conexiones automáticamente.
- Usa **parámetros** para evitar inyecciones SQL.
- Maneja errores con bloques `try-catch`.
- Reutiliza conexiones cuando sea posible.
- Para aplicaciones grandes, separa la lógica de acceso a datos en una **capa DAL (Data Access Layer)**.

---

10. Ejercicio final

Enunciado

Crea un programa de consola en C# que permita:

1. Conectarse a una base de datos.
2. Insertar un estudiante.
3. Listar todos los estudiantes.

4. Actualizar la edad de un estudiante.
5. Eliminar un estudiante por nombre.

Solución (resumida)

```
class Estudiante {
    public int Id;
    public string Nombre;
    public int Edad;
    public string Carrera;
}

class ProgramaEstudiantes {
    static string cadenaConexion =
"Server=localhost;Database=Alumnos;Trusted_Connection=True;";

    static void Insertar(Estudiante e) {
        using var con = new SqlConnection(cadenaConexion);
        con.Open();
        string sql = "INSERT INTO Estudiantes (Nombre, Edad, Carrera) VALUES
(@n, @e, @c)";
        using var cmd = new SqlCommand(sql, con);
        cmd.Parameters.AddWithValue("@n", e.Nombre);
        cmd.Parameters.AddWithValue("@e", e.Edad);
        cmd.Parameters.AddWithValue("@c", e.Carrera);
        cmd.ExecuteNonQuery();
    }

    static void Listar() {
        using var con = new SqlConnection(cadenaConexion);
        con.Open();
        string sql = "SELECT * FROM Estudiantes";
        using var cmd = new SqlCommand(sql, con);
        using var lector = cmd.ExecuteReader();
        while (lector.Read())
            Console.WriteLine($"{lector["Id"]}: {lector["Nombre"]} -
{lector["Edad"]} años - {lector["Carrera"]}");
    }
```

```
    // Métodos para Actualizar y Eliminar similares...


    static void Main() {
        Insertar(new Estudiante { Nombre = "Ana", Edad = 20, Carrera = "Biología" });
        Listar();
    }
}
```

Conclusión

ADO.NET sigue siendo una herramienta muy útil en aplicaciones empresariales y proyectos que requieren control fino del acceso a datos. Es importante comprender su funcionamiento antes de usar herramientas más modernas como Entity Framework.

Recursos adicionales

- Documentación oficial de ADO.NET (Microsoft)
- Tutoriales de SQL Server y C# en YouTube y Microsoft Learn
- Libros recomendados: *C# 10 y .NET 6 – Desarrollo de Aplicaciones* de Mark J. Price.

Parte VII – Seguridad y Buenas Prácticas

Aquí tienes una lección extensa y completa sobre **"Seguridad en Aplicaciones C#"**, ideal para desarrolladores que desean comprender y aplicar buenas prácticas de seguridad al programar con C# en entornos .NET. Esta lección incluye teoría, ejemplos prácticos, y ejercicios con soluciones.

Lección 28: Seguridad en Aplicaciones C#

Objetivos de la lección

- Comprender los principales riesgos de seguridad en aplicaciones C#.
- Aplicar buenas prácticas para proteger datos, usuarios y recursos.
- Conocer técnicas de cifrado, validación, autenticación y autorización.
- Detectar vulnerabilidades comunes y cómo evitarlas.
- Implementar seguridad en entornos de escritorio y web en .NET.

---

1. Fundamentos de Seguridad en Aplicaciones

1.1. Principios de Seguridad

- **Confidencialidad:** proteger la información de accesos no autorizados.
- **Integridad:** garantizar que los datos no sean alterados.
- **Disponibilidad:** asegurar que los servicios estén disponibles.
- **Autenticación:** verificar la identidad del usuario.
- **Autorización:** determinar qué acciones puede realizar el usuario.
- **Auditoría:** registrar eventos para detectar usos indebidos.

---

2. Riesgos Comunes en Aplicaciones C#

1. **Inyecciones SQL**
2. **Uso inseguro de cadenas de conexión**
3. **Manejo inadecuado de contraseñas**
4. **Falta de validación de entrada**
5. **Exposición de información sensible en excepciones**
6. **Inyecciones de código o scripts (Web)**

3. Protección de Datos Sensibles

3.1. Uso de `SecureString` (solo para memoria)

```
using System.Security;

SecureString password = new SecureString();
foreach (char c in "miContraseña123")
    password.AppendChar(c);
```

3.2. Cifrado de Datos

A. Cifrado Simétrico (AES)

```
using System.Security.Cryptography;
using System.Text;

public static class Seguridad
{
    public static byte[] EncriptarAES(string texto, byte[] clave, byte[] iv)
    {
        using Aes aes = Aes.Create();
        aes.Key = clave;
        aes.IV = iv;

        ICryptoTransform encryptor = aes.CreateEncryptor();
        byte[] datos = Encoding.UTF8.GetBytes(texto);
        return encryptor.TransformFinalBlock(datos, 0, datos.Length);
    }
}
```

B. Cifrado Hash (SHA256)

```
using System.Security.Cryptography;
using System.Text;

public static string ObtenerSHA256(string input)
{
    using SHA256 sha256 = SHA256.Create();
    byte[] bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(input));
    return Convert.ToBase64String(bytes);
}
```

---

4. Validación de Entradas

4.1. Ejemplo de validación de entrada

```
public bool ValidarNombreUsuario(string nombre)
{
    return Regex.IsMatch(nombre, @"^[a-zA-Z0-9_]{3,16}$");
}
```

4.2. Evitar Inyecciones SQL

Mal:

```
string query = $"SELECT * FROM Usuarios WHERE nombre='{usuario}'";
```

Bien (con parámetros):

```
using SqlCommand cmd = new SqlCommand("SELECT * FROM Usuarios WHERE
nombre=@nombre", conexion);
cmd.Parameters.AddWithValue("@nombre", usuario);
```

---

5. Autenticación y Autorización

5.1. Autenticación Básica

```
public bool Autenticar(string usuario, string password)
{
    string hashGuardado = ObtenerHashDeDB(usuario);
    return ObtenerSHA256(password) == hashGuardado;
}
```

5.2. Roles y permisos (básico)

```
public enum Rol { Admin, Usuario, Invitado }

public bool TienePermiso(Rol rol, string accion)
{
    if (rol == Rol.Admin) return true;
    if (rol == Rol.Usuario && accion != "Eliminar") return true;
    return false;
}
```

---

6. Seguridad en Aplicaciones Web con ASP.NET

6.1. Prevención de XSS

```
<!-- NUNCA insertar texto del usuario sin codificar -->
<span>@Html.Encode(Model.MensajeUsuario)</span>
```

6.2. Prevención de CSRF

- ASP.NET MVC y Razor Pages usan `@Html.AntiForgeryToken()` automáticamente.

6.3. Autenticación en ASP.NET Core

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
        .AddCookie();
```

---

7. Registro y Auditoría

7.1. Ejemplo simple de logging

```
public void RegistrarAcceso(string usuario)
{
    File.AppendAllText("log.txt", $"{DateTime.Now}: Acceso de {usuario}\n");
}
```

Mejor: usar `ILogger` en .NET Core.

---

8. Ejercicios Prácticos

Ejercicio 1: Validar nombres de usuario

Enunciado:
Escribe una función que valide si un nombre de usuario solo contiene letras, números y guiones bajos, y tiene entre 3 y 16 caracteres.

Solución:

```
public bool ValidarUsuario(string nombre)
{
    return Regex.IsMatch(nombre, @"^[a-zA-Z0-9_]{3,16}$");
}
```

Ejercicio 2: Cifrar una contraseña con SHA256

Enunciado:
Escribe una función que devuelva el hash SHA256 de una contraseña ingresada por el usuario.

Solución:

```
public string HashPassword(string password)
{
    using SHA256 sha = SHA256.Create();
    byte[] hashBytes = sha.ComputeHash(Encoding.UTF8.GetBytes(password));
    return Convert.ToBase64String(hashBytes);
}
```

Ejercicio 3: SQL Seguro con parámetros

Enunciado:
Modifica el siguiente código para prevenir inyección SQL.

```
string sql = $"SELECT * FROM clientes WHERE nombre='{nombre}'";
```

Solución:

```
string sql = "SELECT * FROM clientes WHERE nombre = @nombre";
using SqlCommand cmd = new SqlCommand(sql, conexion);
cmd.Parameters.AddWithValue("@nombre", nombre);
```

Ejercicio 4: Rol y autorización

Enunciado:
Crea un sistema simple que controle si un usuario con rol `Admin` puede acceder a una función llamada `EliminarRegistro`.

Solución:

```
public bool PuedeEliminar(string rol)
{
    return rol == "Admin";
}
```

---

9. Buenas Prácticas Generales

- Nunca guardar contraseñas en texto plano.
- Evitar mostrar trazas de error al usuario final.
- Cerrar y disponer correctamente las conexiones.
- Usar librerías oficiales para criptografía (`System.Security.Cryptography`).
- No reutilizar claves ni vectores IV.
- Usar variables `readonly` o `const` para valores sensibles.

Recursos Recomendados

- OWASP Top 10
- Documentación oficial: .NET Security
- Libros: "C# 12 and .NET 8 – Modern Cross-Platform Development"

Lección Completa 29: Buenas Prácticas de Programación en C#

Objetivos de la lección

- Entender qué son las buenas prácticas de programación.
- Aplicar principios de programación limpia (Clean Code).
- Aprender convenciones y estilos de codificación en C#.
- Evitar errores comunes y prácticas perjudiciales.
- Escribir código legible, reutilizable, y fácil de mantener.

---

1. ¿Qué son las buenas prácticas?

Las **buenas prácticas de programación** son un conjunto de recomendaciones y principios que mejoran la calidad del software. Se enfocan en aspectos como:

- Legibilidad
- Mantenibilidad
- Reutilización
- Eficiencia
- Seguridad

Estas prácticas permiten que cualquier desarrollador (incluyéndote en el futuro) entienda y trabaje con el código sin dificultad.

---

2. Nombres Significativos

Buena práctica:

Usar nombres descriptivos para variables, métodos, clases y archivos.

```
int numeroDeEstudiantes; // Claro y preciso

string nombreCompleto;   // Nombre significativo
```

Mala práctica:

```
int n;   // No dice nada

string x; // ¿Qué es "x"?
```

Reglas:

- Usa **camelCase** para variables y parámetros: `numeroTotal`
- Usa **PascalCase** para métodos y clases: `CalcularPromedio`, `Estudiante`

- Evita abreviaciones salvo que sean muy comunes: `id`, `URL`, `HTML`

---

3. Principios SOLID

El conjunto de principios **SOLID** mejora la arquitectura de tu código.

S — Single Responsibility Principle (Responsabilidad Única)

Una clase debe tener una sola razón para cambiar.

```
//  Buena práctica
class Reporte {
    public void Generar() { ... }
}


class Impresora {
    public void Imprimir() { ... }
}
```

O — Open/Closed Principle (Abierto/Cerrado)

El código debe estar **abierto a la extensión**, pero **cerrado a la modificación**.

L — Liskov Substitution Principle

Las clases hijas deben poder usarse sin alterar el comportamiento de la clase base.

I — Interface Segregation Principle

No forzar a una clase a implementar interfaces que no usa.

D — Dependency Inversion Principle

Depender de abstracciones, no de clases concretas.

---

4. Comentarios útiles

Comentarios útiles:

- Explican el **por qué**, no el **qué**
- Se usan para marcar TODOs, referencias o ideas complejas

```
// Calculamos el promedio ponderado porque hay notas con distinto peso
double promedio = (nota1 * 0.3) + (nota2 * 0.7);
```

Comentarios inútiles:

```
// Suma dos números
int suma = a + b;  // ¡Obvio!
```

---

5. Código limpio y organizado

Organiza el código:

- Una clase por archivo.
- Métodos cortos y que hagan una sola cosa.
- Usa espacios en blanco para separar bloques lógicos.

```
class Estudiante {
    public string Nombre { get; set; }

    public double CalcularPromedio(double[] notas) {
        return notas.Average();
    }
}
```

Evita:

- Métodos largos.
- Código duplicado.
- Variables globales innecesarias.

---

6. Manejo de errores

Usa `try-catch` solo cuando sea necesario.

```
try {
    var resultado = int.Parse("abc");
} catch (FormatException ex) {
    Console.WriteLine("Error de formato: " + ex.Message);
}
```

No atrapes excepciones genéricas sin control:

```
catch (Exception e) {
    // Mal: no sabemos qué pasó
}
```

---

7. Validación y defensiva

Siempre valida entradas, especialmente si vienen del usuario.

```
public void RegistrarEdad(int edad) {
    if (edad < 0 || edad > 120) {
```

```
        throw new ArgumentOutOfRangeException("Edad inválida.");
    }
}
```

---

8. Estilo y convenciones

Espaciado y sangría:

```
if (condicion) {
    // Bien indentado
    Ejecutar();
}
```

Llaves:

Siempre usa llaves, incluso para una sola línea.

```
if (condicion) {
    Ejecutar();
}
```

---

9. DRY vs. WET

- **DRY (Don't Repeat Yourself):** evita repetir código
- **WET (Write Everything Twice):** mala práctica

Repetido:

```
Console.WriteLine("Bienvenido");
Console.WriteLine("Bienvenido");
```

Función reutilizable:

```
void MostrarMensajeBienvenida() {
    Console.WriteLine("Bienvenido");
}
```

---

10. Buenas prácticas con colecciones y LINQ

Usa `foreach`, `LINQ`, y `var` correctamente:

```
foreach (var alumno in listaAlumnos) {
    Console.WriteLine(alumno.Nombre);
}

// LINQ para filtrar
```

```
var mayores = listaAlumnos.Where(a => a.Edad >= 18);
```

---

11. Separación de responsabilidades

- Presentación (UI)
- Lógica de negocio
- Acceso a datos

Usa **capas** para separar responsabilidades. No mezcles interfaz con lógica.

---

12. Testing y pruebas

Siempre que puedas, **escribe pruebas unitarias**.

```
[TestMethod]
public void TestSuma() {
    Assert.AreEqual(5, Calculadora.Sumar(2, 3));
}
```

---

13. Buenas herramientas

- **Visual Studio / Rider**
- **Resharper** para análisis de código
- **SonarLint** para detectar errores y malos olores

🛠️ 14. Resumen de Reglas de Oro

Regla	Descripción
Nombres claros	Usa nombres que indiquen la intención
Métodos cortos	Haz una sola cosa por método
Sin duplicación	Reutiliza funciones y estructuras
Código legible	Usa espacios, comentarios útiles
Usa excepciones correctamente	Maneja errores de forma clara
Aplica SOLID	Mejora estructura y escalabilidad
Valida entradas	Sé defensivo ante errores de usuario

Ejercicio Final

Reescribe el siguiente código aplicando buenas prácticas:

```
class A {
    public void m() {
        int x = 0;
```

```
        for (int i = 0; i < 10; i++) { x = x + i; }
        Console.WriteLine(x);
    }
}
```

✔ Sugerencia de solución:

```
class Calculadora {
    public int CalcularSumaDelCeroAlNueve() {
        int suma = 0;
        for (int i = 0; i < 10; i++) {
            suma += i;
        }
        return suma;
    }

    public void MostrarResultado() {
        int resultado = CalcularSumaDelCeroAlNueve();
        Console.WriteLine(resultado);
    }
}
```

Parte VIII – Proyectos y Aplicaciones Reales

Lección 30: Proyecto 1 – Sistema de Gestión de Biblioteca

Tecnologías:

- C# (Windows Forms)
- SQL Server (base de datos local o remota)
- ADO.NET (acceso a datos)

Objetivos del Proyecto

Crear una aplicación de escritorio en Windows Forms que permita:

- Gestionar libros (agregar, editar, eliminar, buscar)
- Gestionar usuarios o socios
- Registrar préstamos y devoluciones de libros
- Visualizar libros disponibles y en préstamo

Requisitos Previos

- Visual Studio (preferiblemente 2022 o superior)
- SQL Server (puede ser SQL Server Express)
- Conocimientos básicos de:
    - C#
    - SQL
    - Windows Forms
    - CRUD (Create, Read, Update, Delete)

Estructura del Proyecto

- **Forms**:
    - `MainForm`: pantalla principal con menú de navegación
    - `FormLibros`: gestión de libros
    - `FormUsuarios`: gestión de socios

- `FormPrestamos`: gestión de préstamos
- **Clases**:
  - `Libro`, `Usuario`, `Prestamo`
  - `ConexionBD` (gestiona conexión a SQL Server)
- **Base de Datos**:
  - Tablas: `Libros`, `Usuarios`, `Prestamos`

Paso 1: Crear la Base de Datos en SQL Server

Script SQL:

```
CREATE DATABASE BibliotecaDB;
GO

USE BibliotecaDB;

CREATE TABLE Libros (
    Id INT PRIMARY KEY IDENTITY,
    Titulo NVARCHAR(100),
    Autor NVARCHAR(100),
    ISBN NVARCHAR(20),
    Disponible BIT
);

CREATE TABLE Usuarios (
    Id INT PRIMARY KEY IDENTITY,
    Nombre NVARCHAR(100),
    Email NVARCHAR(100)
);

CREATE TABLE Prestamos (
    Id INT PRIMARY KEY IDENTITY,
    LibroId INT FOREIGN KEY REFERENCES Libros(Id),
    UsuarioId INT FOREIGN KEY REFERENCES Usuarios(Id),
    FechaPrestamo DATE,
    FechaDevolucion DATE
);
```

Paso 2: Crear el Proyecto en Visual Studio

1. Abrir Visual Studio → "Crear nuevo proyecto"
2. Elegir **Aplicación de Windows Forms (.NET Framework)**
3. Nombre: `SistemaBiblioteca`
4. En el **Explorador de soluciones**, agregar tres nuevos formularios:
   - `FormLibros.cs`
   - `FormUsuarios.cs`
   - `FormPrestamos.cs`

Paso 3: Conectar a la Base de Datos

Clase `ConexionBD.cs`:

```
using System.Data.SqlClient;


public class ConexionBD
{
    private static string cadena =
"Server=.;Database=BibliotecaDB;Trusted_Connection=True;";


    public static SqlConnection ObtenerConexion()
    {
        return new SqlConnection(cadena);
    }
}
```

Paso 4: CRUD de Libros

Clase `Libro.cs`:

```
public class Libro
{
    public int Id { get; set; }
    public string Titulo { get; set; }
    public string Autor { get; set; }
    public string ISBN { get; set; }
    public bool Disponible { get; set; }
```

```
}
```

Diseño de `FormLibros`:

- TextBoxes: `txtTitulo`, `txtAutor`, `txtISBN`
- CheckBox: `chkDisponible`
- DataGridView: `dgvLibros`
- Botones: `btnAgregar`, `btnEditar`, `btnEliminar`, `btnBuscar`

Lógica para Agregar Libro:

```
private void btnAgregar_Click(object sender, EventArgs e)
{
    using (SqlConnection con = ConexionBD.ObtenerConexion())
    {
        con.Open();
        string query = "INSERT INTO Libros (Titulo, Autor, ISBN, Disponible) VALUES (@Titulo, @Autor, @ISBN, @Disponible)";
        SqlCommand cmd = new SqlCommand(query, con);
        cmd.Parameters.AddWithValue("@Titulo", txtTitulo.Text);
        cmd.Parameters.AddWithValue("@Autor", txtAutor.Text);
        cmd.Parameters.AddWithValue("@ISBN", txtISBN.Text);
        cmd.Parameters.AddWithValue("@Disponible", chkDisponible.Checked);
        cmd.ExecuteNonQuery();
        MessageBox.Show("Libro agregado correctamente.");
        MostrarLibros();
    }
}
```

Método `MostrarLibros()`:

```
private void MostrarLibros()
{
    using (SqlConnection con = ConexionBD.ObtenerConexion())
    {
        SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Libros", con);
        DataTable dt = new DataTable();
        da.Fill(dt);
        dgvLibros.DataSource = dt;
    }
}
```

Paso 5: CRUD de Usuarios

Similar a libros:

- Clase: `Usuario.cs`
- Formulario: `FormUsuarios.cs`
- Controles: `txtNombre`, `txtEmail`, `dgvUsuarios`, botones CRUD
- SQL: usar tabla `Usuarios`

Paso 6: Gestión de Préstamos

Diseño de `FormPrestamos.cs`:

- Combos: `cmbLibros`, `cmbUsuarios`
- DatePickers: `dtpPrestamo`, `dtpDevolucion`
- Botón: `btnRegistrar`

Código para Registrar Préstamo:

```
private void btnRegistrar_Click(object sender, EventArgs e)
{
    using (SqlConnection con = ConexionBD.ObtenerConexion())
    {
        con.Open();

        string query = "INSERT INTO Prestamos (LibroId, UsuarioId, FechaPrestamo, FechaDevolucion) VALUES (@LibroId, @UsuarioId, @FechaPrestamo, @FechaDevolucion)";
        SqlCommand cmd = new SqlCommand(query, con);
        cmd.Parameters.AddWithValue("@LibroId", (int)cmbLibros.SelectedValue);
        cmd.Parameters.AddWithValue("@UsuarioId", (int)cmbUsuarios.SelectedValue);
        cmd.Parameters.AddWithValue("@FechaPrestamo", dtpPrestamo.Value);
        cmd.Parameters.AddWithValue("@FechaDevolucion", dtpDevolucion.Value);
        cmd.ExecuteNonQuery();

        // Marcar libro como no disponible
        string update = "UPDATE Libros SET Disponible = 0 WHERE Id = @LibroId";
        SqlCommand cmdUpdate = new SqlCommand(update, con);
```

```
        cmdUpdate.Parameters.AddWithValue("@LibroId",
(int)cmbLibros.SelectedValue);
        cmdUpdate.ExecuteNonQuery();


        MessageBox.Show("Préstamo registrado.");
    }
}
```

Paso 7: Crear Menú Principal

`MainForm.cs`

- Menú con opciones: Libros, Usuarios, Préstamos
- Cada opción abre su formulario respectivo

```
private void librosToolStripMenuItem_Click(object sender, EventArgs e)
{
    FormLibros fl = new FormLibros();
    fl.ShowDialog();
}
```

Paso 8: Pruebas y Validaciones

Agregar:

- Validación de campos vacíos
- Mensajes de confirmación
- Comprobación de disponibilidad antes de préstamo
- Vistas para libros disponibles o en préstamo

Paso 9: Mejoras Futuras

- Filtro por título o autor
- Registro de historial de préstamos por usuario
- Reportes en PDF o Excel
- Exportar/Importar datos
- Interfaz moderna con Guna UI o Bunifu

Conclusión

Este proyecto te enseña a:

- Crear una arquitectura básica para aplicaciones CRUD en Windows Forms
- Usar ADO.NET para conectar a SQL Server
- Gestionar datos con formularios intuitivos
- Aplicar lógica de negocios básica (como préstamos y devoluciones)

Lección 31: Proyecto 2 – Aplicación Web de Blog Personal (ASP.NET Core MVC + EF Core)

Objetivos de Aprendizaje

Al finalizar esta lección, podrás:

- Crear una aplicación web funcional tipo blog personal.
- Utilizar ASP.NET Core MVC para la estructura del sitio.
- Conectar con una base de datos usando Entity Framework Core.
- Implementar operaciones CRUD (crear, leer, actualizar, eliminar) para entradas del blog.
- Usar migraciones y trabajar con una base de datos local.
- Aplicar un diseño limpio y funcional usando Razor Views.

Tecnologías Usadas

- **Lenguaje**: C#
- **Framework Web**: ASP.NET Core MVC
- **ORM**: Entity Framework Core
- **Base de Datos**: SQLite (o SQL Server local)
- **IDE recomendado**: Visual Studio o Visual Studio Code

Estructura del Proyecto

```
BlogPersonal/
│
├── Controllers/
│   └── PostsController.cs
│
├── Models/
│   └── Post.cs
│   └── BlogContext.cs
│
```

```
├── Views/
│   ├── Posts/
│   │   └── Index.cshtml
│   │   └── Create.cshtml
│   │   └── Edit.cshtml
│   │   └── Details.cshtml
│   │   └── Delete.cshtml
│
├── wwwroot/
│   └── css/, js/, img/
│
├── Program.cs
├── Startup.cs (si es versión .NET Core 3.1/5.0)
└── appsettings.json
```

Paso 1: Crear el Proyecto

Abre la terminal o Visual Studio y ejecuta:

```
dotnet new mvc -n BlogPersonal
cd BlogPersonal
```

Agrega EF Core y proveedor SQLite:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

Paso 2: Crear el Modelo `Post`

Archivo: `Models/Post.cs`

```
using System;
using System.ComponentModel.DataAnnotations;

namespace BlogPersonal.Models
{
    public class Post
    {
        public int Id { get; set; }

        [Required, StringLength(100)]
```

```
        public string Titulo { get; set; }

        [Required]
        public string Contenido { get; set; }

        public DateTime FechaPublicacion { get; set; } = DateTime.Now;
    }
}
```

Paso 3: Crear el Contexto `BlogContext`

Archivo: `Models/BlogContext.cs`

```
using Microsoft.EntityFrameworkCore;

namespace BlogPersonal.Models
{
    public class BlogContext : DbContext
    {
        public BlogContext(DbContextOptions<BlogContext> options)
            : base(options)
        {
        }

        public DbSet<Post> Posts { get; set; }
    }
}
```

Paso 4: Configurar EF Core en `Program.cs`

Si usas .NET 6 o superior:

```
builder.Services.AddDbContext<BlogContext>(options =>
    options.UseSqlite("Data Source=blog.db"));
```

No olvides importar el espacio de nombres:

```
using BlogPersonal.Models;
using Microsoft.EntityFrameworkCore;
```

Paso 5: Crear la Base de Datos

En terminal:

```
dotnet ef migrations add InitialCreate

dotnet ef database update
```

Esto genera `blog.db`.

Paso 6: Crear el Controlador `PostsController`

Archivo: `Controllers/PostsController.cs`

```
using Microsoft.AspNetCore.Mvc;

using BlogPersonal.Models;

using System.Linq;


namespace BlogPersonal.Controllers

{

    public class PostsController : Controller

    {

        private readonly BlogContext _context;


        public PostsController(BlogContext context)

        {

            _context = context;

        }


        public IActionResult Index() => View(_context.Posts.OrderByDescending(p
=> p.FechaPublicacion).ToList());


        public IActionResult Details(int id)

        {

            var post = _context.Posts.Find(id);

            if (post == null) return NotFound();

            return View(post);

        }


        public IActionResult Create() => View();
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(Post post)
{
    if (ModelState.IsValid)
    {
        _context.Add(post);
        _context.SaveChanges();
        return RedirectToAction(nameof(Index));
    }
    return View(post);
}

public IActionResult Edit(int id)
{
    var post = _context.Posts.Find(id);
    if (post == null) return NotFound();
    return View(post);
}

[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(int id, Post post)
{
    if (id != post.Id) return NotFound();

    if (ModelState.IsValid)
    {
        _context.Update(post);
        _context.SaveChanges();
        return RedirectToAction(nameof(Index));
    }
    return View(post);
}

public IActionResult Delete(int id)
```

```
        {
            var post = _context.Posts.Find(id);
            if (post == null) return NotFound();
            return View(post);
        }

        [HttpPost, ActionName("Delete")]
        [ValidateAntiForgeryToken]
        public IActionResult DeleteConfirmed(int id)
        {
            var post = _context.Posts.Find(id);
            _context.Posts.Remove(post);
            _context.SaveChanges();
            return RedirectToAction(nameof(Index));
        }
    }
}
```

Paso 7: Crear las Vistas (Razor)

Ejemplo: `Views/Posts/Index.cshtml`

```
@model IEnumerable<BlogPersonal.Models.Post>

<h2>Mis Publicaciones</h2>

<a asp-action="Create">Crear nueva entrada</a>

<ul>
@foreach (var post in Model)
{
    <li>
        <strong>@post.Titulo</strong>
(@post.FechaPublicacion.ToShortDateString()) |
        <a asp-action="Details" asp-route-id="@post.Id">Ver</a> |
        <a asp-action="Edit" asp-route-id="@post.Id">Editar</a> |
        <a asp-action="Delete" asp-route-id="@post.Id">Eliminar</a>
    </li>
```

```
}
</ul>
```

Crea vistas similares para `Create`, `Edit`, `Details`, y `Delete` usando plantillas básicas de Razor.

Opcional: Estilos CSS

Coloca un archivo en `wwwroot/css/site.css` y enlázalo en `_Layout.cshtml`.

Paso 8: Ejecutar la Aplicación

```
dotnet run
```

Abre el navegador en `https://localhost:5001/Posts`.

Sugerencias de Mejoras

- Agregar validación en cliente con jQuery.
- Implementar autenticación de usuario (Identity).
- Agregar categorías o etiquetas.
- Agregar un sistema de comentarios.
- Crear una interfaz pública/lectura y otra privada/admin.

Ejercicio Final

Modifica el blog para que cada entrada pueda incluir una imagen. Para ello:

1. Agrega una propiedad `string ImagenUrl` al modelo.
2. Modifica las vistas para mostrar y subir la imagen.
3. Almacena las imágenes en `wwwroot/images/`.

Lección 32: App de Notas con Serialización (C# + JSON + Windows Forms)

Objetivo

Crear una aplicación de escritorio con **Windows Forms** en C# que permita al usuario agregar, editar, eliminar y guardar notas. Las notas se almacenarán en un archivo JSON mediante serialización y deserialización para conservar los datos entre sesiones.

Contenidos

1. Conceptos clave
2. Preparación del entorno
3. Diseño de la interfaz gráfica (Windows Forms)
4. Modelo de datos: clase Nota
5. Serialización y deserialización JSON
6. Implementación funcional
7. Guardar y cargar notas automáticamente
8. Mejoras y recomendaciones

---

1. Conceptos clave

¿Qué es serialización?

La serialización es el proceso de convertir un objeto en un formato que pueda almacenarse o transmitirse (como JSON, XML, binario). La deserialización es el proceso inverso, convertir ese formato en un objeto.

¿Por qué usar JSON?

- JSON es un formato de texto muy usado.
- Es fácil de leer y escribir.
- Está bien soportado en C# con bibliotecas como `System.Text.Json` o `Newtonsoft.Json`.

Windows Forms

- Plataforma para crear interfaces gráficas de escritorio en C#.

- Usa controles como botones, listas, cuadros de texto, etc.
- Fácil para prototipos y aplicaciones pequeñas.

---

2. Preparación del entorno

- IDE recomendado: **Visual Studio** (Community o superior).
- Crear un nuevo proyecto:
  - Tipo: Windows Forms App (.NET Framework o .NET 6/7/8 según tu preferencia).
  - Lenguaje: C#
- Asegúrate de tener instalado el paquete **System.Text.Json** si usas .NET Core o superior (normalmente ya viene incluido).
- Alternativamente, puedes usar **Newtonsoft.Json** (instalable desde NuGet).

---

3. Diseño de la interfaz gráfica

Vamos a necesitar:

- **ListBox** para mostrar la lista de notas.
- **TextBox** para editar el contenido de la nota.
- **Button** para agregar nueva nota.
- **Button** para eliminar nota.
- **Button** para guardar cambios.

Diseño sugerido:

Control	Nombre	Propósito
ListBox	lstNotas	Mostrar títulos o listas de notas
TextBox Multilínea	txtContenido	Editar el contenido de la nota
Button	btnAgregar	Agregar una nueva nota
Button	btnEliminar	Eliminar la nota seleccionada
Button	btnGuardar	Guardar los cambios en archivo

---

4. Modelo de datos: Clase Nota

Creamos una clase simple para representar cada nota:

```
public class Nota
{
    public string Titulo { get; set; }
    public string Contenido { get; set; }
    public DateTime FechaCreacion { get; set; }
```

```
    public Nota()
    {
        FechaCreacion = DateTime.Now;
    }
}
```

---

5. Serialización y deserialización JSON

Usando `System.Text.Json` (C# .NET Core/5+):

```
using System.Text.Json;
using System.Text.Json.Serialization;

// Serializar lista de notas a JSON
string json = JsonSerializer.Serialize(listaNotas);

// Deserializar JSON a lista de notas
List<Nota> listaNotas = JsonSerializer.Deserialize<List<Nota>>(json);
```

Guardar y cargar desde archivo:

```
string rutaArchivo = "notas.json";

// Guardar
File.WriteAllText(rutaArchivo, json);

// Cargar
if(File.Exists(rutaArchivo))
{
    string jsonLeido = File.ReadAllText(rutaArchivo);
    listaNotas = JsonSerializer.Deserialize<List<Nota>>(jsonLeido);
}
else
{
    listaNotas = new List<Nota>();
}
```

---

6. Implementación funcional paso a paso

Variables globales

```
List<Nota> listaNotas = new List<Nota>();

string rutaArchivo = "notas.json";
```

Al cargar el formulario:

```
private void Form1_Load(object sender, EventArgs e)

{

    CargarNotas();

    ActualizarLista();

}
```

Método para cargar notas

```
private void CargarNotas()

{

    if (File.Exists(rutaArchivo))

    {

        string jsonLeido = File.ReadAllText(rutaArchivo);

        listaNotas = JsonSerializer.Deserialize<List<Nota>>(jsonLeido);

    }

    else

    {

        listaNotas = new List<Nota>();

    }

}
```

Método para guardar notas

```
private void GuardarNotas()

{

    string json = JsonSerializer.Serialize(listaNotas, new JsonSerializerOptions
{ WriteIndented = true });

    File.WriteAllText(rutaArchivo, json);

}
```

Actualizar ListBox con títulos de notas

```
private void ActualizarLista()

{

    lstNotas.Items.Clear();

    foreach(var nota in listaNotas)
```

```
    {
        lstNotas.Items.Add(nota.Titulo);
    }
}
```

Evento para agregar nota

```
private void btnAgregar_Click(object sender, EventArgs e)
{
    Nota nuevaNota = new Nota { Titulo = "Nueva Nota", Contenido = "" };
    listaNotas.Add(nuevaNota);
    ActualizarLista();
    lstNotas.SelectedIndex = listaNotas.Count - 1; // Seleccionar la nueva nota
}
```

Evento para eliminar nota

```
private void btnEliminar_Click(object sender, EventArgs e)
{
    int index = lstNotas.SelectedIndex;
    if(index >= 0)
    {
        listaNotas.RemoveAt(index);
        ActualizarLista();
        txtContenido.Clear();
    }
}
```

Evento para guardar contenido modificado

```
private void btnGuardar_Click(object sender, EventArgs e)
{
    int index = lstNotas.SelectedIndex;
    if (index >= 0)
    {
        listaNotas[index].Contenido = txtContenido.Text;
        listaNotas[index].Titulo =
ObtenerTituloDesdeContenido(txtContenido.Text);
        ActualizarLista();
        GuardarNotas();
    }
}
```

```
private string ObtenerTituloDesdeContenido(string contenido)
{
    if (string.IsNullOrWhiteSpace(contenido))
        return "Nota vacía";

    // Obtener primera línea o primeros 20 caracteres como título
    var lineas = contenido.Split(new[] { '\r', '\n' },
StringSplitOptions.RemoveEmptyEntries);
    string titulo = lineas.Length > 0 ? lineas[0] : contenido;
    return titulo.Length > 20 ? titulo.Substring(0, 20) + "..." : titulo;
}
```

Evento para mostrar contenido al seleccionar nota

```
private void lstNotas_SelectedIndexChanged(object sender, EventArgs e)
{
    int index = lstNotas.SelectedIndex;
    if(index >= 0)
    {
        txtContenido.Text = listaNotas[index].Contenido;
    }
    else
    {
        txtContenido.Clear();
    }
}
```

---

7. Guardar y cargar automáticamente

Para mejorar la experiencia, puedes guardar automáticamente al modificar el texto o al cambiar la selección.

Ejemplo guardado al perder foco en el TextBox:

```
private void txtContenido_Leave(object sender, EventArgs e)
{
    int index = lstNotas.SelectedIndex;
    if (index >= 0)
    {
        listaNotas[index].Contenido = txtContenido.Text;
```

```
        listaNotas[index].Titulo =
ObtenerTituloDesdeContenido(txtContenido.Text);

        ActualizarLista();

        GuardarNotas();

    }

}
```

---

8. Mejoras y recomendaciones

- **Validar entrada:** Evitar notas con títulos vacíos.
- **Agregar búsqueda:** Filtrar notas en ListBox por texto.
- **Ordenar notas:** Por fecha de creación o alfabéticamente.
- **Backup:** Guardar copia del archivo JSON para evitar pérdidas.
- **UI amigable:** Agregar iconos, colores o categorías a las notas.
- **Multihilo:** Usar tareas para guardar sin bloquear UI en archivos grandes.

Código completo (simplificado)

```
using System;

using System.Collections.Generic;

using System.IO;

using System.Text.Json;

using System.Windows.Forms;


namespace AppNotas

{

    public partial class Form1 : Form

    {

        List<Nota> listaNotas = new List<Nota>();

        string rutaArchivo = "notas.json";


        public Form1()

        {

            InitializeComponent();

            Load += Form1_Load;

            lstNotas.SelectedIndexChanged += lstNotas_SelectedIndexChanged;

            btnAgregar.Click += btnAgregar_Click;
```

```
            btnEliminar.Click += btnEliminar_Click;
            btnGuardar.Click += btnGuardar_Click;
            txtContenido.Leave += txtContenido_Leave;
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            CargarNotas();
            ActualizarLista();
        }

        private void CargarNotas()
        {
            if (File.Exists(rutaArchivo))
            {
                string jsonLeido = File.ReadAllText(rutaArchivo);
                listaNotas = JsonSerializer.Deserialize<List<Nota>>(jsonLeido);
            }
            else
            {
                listaNotas = new List<Nota>();
            }
        }

        private void GuardarNotas()
        {
            string json = JsonSerializer.Serialize(listaNotas, new
JsonSerializerOptions { WriteIndented = true });
            File.WriteAllText(rutaArchivo, json);
        }

        private void ActualizarLista()
        {
            lstNotas.Items.Clear();
            foreach (var nota in listaNotas)
            {
                lstNotas.Items.Add(nota.Titulo);
```

```
    }
}

private void btnAgregar_Click(object sender, EventArgs e)
{
    Nota nuevaNota = new Nota { Titulo = "Nueva Nota", Contenido = "" };
    listaNotas.Add(nuevaNota);
    ActualizarLista();
    lstNotas.SelectedIndex = listaNotas.Count - 1;
}

private void btnEliminar_Click(object sender, EventArgs e)
{
    int index = lstNotas.SelectedIndex;
    if (index >= 0)
    {
        listaNotas.RemoveAt(index);
        ActualizarLista();
        txtContenido.Clear();
    }
}

private void btnGuardar_Click(object sender, EventArgs e)
{
    GuardarNotaSeleccionada();
}

private void txtContenido_Leave(object sender, EventArgs e)
{
    GuardarNotaSeleccionada();
}

private void GuardarNotaSeleccionada()
{
    int index = lstNotas.SelectedIndex;
    if (index >= 0)
```

```
            {
                listaNotas[index].Contenido = txtContenido.Text;
                listaNotas[index].Titulo =
ObtenerTituloDesdeContenido(txtContenido.Text);
                ActualizarLista();
                GuardarNotas();
            }
        }

        private void lstNotas_SelectedIndexChanged(object sender, EventArgs e)
        {
            int index = lstNotas.SelectedIndex;
            if (index >= 0)
            {
                txtContenido.Text = listaNotas[index].Contenido;
            }
            else
            {
                txtContenido.Clear();
            }
        }

        private string ObtenerTituloDesdeContenido(string contenido)
        {
            if (string.IsNullOrWhiteSpace(contenido))
                return "Nota vacía";

            var lineas = contenido.Split(new[] { '\r', '\n' },
StringSplitOptions.RemoveEmptyEntries);
            string titulo = lineas.Length > 0 ? lineas[0] : contenido;
            return titulo.Length > 20 ? titulo.Substring(0, 20) + "..." :
titulo;
        }
    }

    public class Nota
    {
```

```
        public string Titulo { get; set; }
        public string Contenido { get; set; }
        public DateTime FechaCreacion { get; set; }

        public Nota()
        {
            FechaCreacion = DateTime.Now;
        }
    }
}
```

Lección 33: API RESTful de Productos con ASP.NET Core + Swagger + JWT

Índice

---

1. Introducción

En esta lección vamos a crear una API RESTful que gestione productos. La API soportará operaciones CRUD (Crear, Leer, Actualizar, Eliminar). Usaremos ASP.NET Core para construir la API, Swagger para documentarla y probarla visualmente, y JWT para proteger las rutas con autenticación.

---

2. Requisitos previos

- Visual Studio 2022 o Visual Studio Code
- .NET 6.0 o superior SDK instalado
- Conocimientos básicos de C# y ASP.NET Core
- Postman (opcional, para probar API)

---

3. Crear el proyecto ASP.NET Core Web API

1. Abre Visual Studio.
2. Crea un nuevo proyecto y selecciona **ASP.NET Core Web API**.
3. Nombre: `ProductosApi`.

4. Framework: .NET 6.0 o superior.
5. Desmarca la opción "Use controllers (uncheck Minimal APIs)" para crear controlador clásico.
6. Crear proyecto.

---

4. Definir el modelo Producto

En la carpeta `Models`, crea una clase llamada `Producto.cs`:

```
namespace ProductosApi.Models
{
    public class Producto
    {
        public int Id { get; set; }
        public string Nombre { get; set; }
        public string Descripcion { get; set; }
        public decimal Precio { get; set; }
        public int Stock { get; set; }
    }
}
```

---

5. Crear el contexto de base de datos con Entity Framework Core

Agrega el paquete EF Core:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

Crea la clase `AppDbContext.cs` en una carpeta `Data`:

```
using Microsoft.EntityFrameworkCore;
using ProductosApi.Models;

namespace ProductosApi.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options)
            : base(options)
        {
```

```
        }

        public DbSet<Producto> Productos { get; set; }
    }
}
```

Configura la cadena de conexión en `appsettings.json`:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\
\mssqllocaldb;Database=ProductosDb;Trusted_Connection=True;"
  }
}
```

En `Program.cs` o `Startup.cs` (según versión):

```
builder.Services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnectio
n")));
```

---

6. Crear el controlador Productos (CRUD)

Crea un controlador `ProductosController.cs` en `Controllers`:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using ProductosApi.Data;
using ProductosApi.Models;

namespace ProductosApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductosController : ControllerBase
    {
        private readonly AppDbContext _context;

        public ProductosController(AppDbContext context)
        {
            _context = context;
```

```
        }

        // GET: api/productos
        [HttpGet]
        public async Task<ActionResult<IEnumerable<Producto>>> GetProductos()
        {
            return await _context.Productos.ToListAsync();
        }

        // GET: api/productos/5
        [HttpGet("{id}")]
        public async Task<ActionResult<Producto>> GetProducto(int id)
        {
            var producto = await _context.Productos.FindAsync(id);

            if (producto == null)
                return NotFound();

            return producto;
        }

        // POST: api/productos
        [HttpPost]
        public async Task<ActionResult<Producto>> PostProducto(Producto
producto)
        {
            _context.Productos.Add(producto);
            await _context.SaveChangesAsync();

            return CreatedAtAction(nameof(GetProducto), new { id =
producto.Id }, producto);
        }

        // PUT: api/productos/5
        [HttpPut("{id}")]
        public async Task<IActionResult> PutProducto(int id, Producto producto)
        {
```

```
            if (id != producto.Id)
                return BadRequest();

            _context.Entry(producto).State = EntityState.Modified;

            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!_context.Productos.Any(e => e.Id == id))
                    return NotFound();
                else
                    throw;
            }

            return NoContent();
        }

        // DELETE: api/productos/5
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteProducto(int id)
        {
            var producto = await _context.Productos.FindAsync(id);
            if (producto == null)
                return NotFound();

            _context.Productos.Remove(producto);
            await _context.SaveChangesAsync();

            return NoContent();
        }
    }
}
```

---

7. Agregar Swagger para documentación de API

Agrega paquete:

```
dotnet add package Swashbuckle.AspNetCore
```

En `Program.cs` agrega:

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

---

8. Implementar autenticación JWT

Agrega paquetes:

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
dotnet add package System.IdentityModel.Tokens.Jwt
```

Define en `appsettings.json` la configuración JWT:

```
"Jwt": {
  "Key": "ClaveSuperSecretaMuyLargaParaJWT1234!",
  "Issuer": "ProductosApi",
  "Audience": "ProductosApiUsuarios"
}
```

En `Program.cs`:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
```

```
using Microsoft.IdentityModel.Tokens;
using System.Text;

var key = Encoding.ASCII.GetBytes(builder.Configuration["Jwt:Key"]);

builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = builder.Configuration["Jwt:Issuer"],
        ValidAudience = builder.Configuration["Jwt:Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(key)
    };
});
```

Agrega la autenticación al pipeline:

```
app.UseAuthentication();
app.UseAuthorization();
```

---

9. Proteger endpoints con autorización JWT

En el controlador `ProductosController`, añade `[Authorize]`:

```
using Microsoft.AspNetCore.Authorization;

[Authorize]
[Route("api/[controller]")]
[ApiController]
public class ProductosController : ControllerBase
{
```

```
    // ...
}
```

---

10. Crear un endpoint para login y generación del token JWT

Agrega un controlador `AuthController` para autenticación simple (usuario fijo para demo):

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

namespace ProductosApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AuthController : ControllerBase
    {
        private readonly IConfiguration _configuration;

        public AuthController(IConfiguration configuration)
        {
            _configuration = configuration;
        }

        [HttpPost("login")]
        public IActionResult Login([FromBody] LoginModel login)
        {
            // Usuario demo fijo (en real, consultar base de datos)
            if (login.Username == "admin" && login.Password == "password")
            {
                var tokenHandler = new JwtSecurityTokenHandler();
                var key = Encoding.ASCII.GetBytes(_configuration["Jwt:Key"]);

                var tokenDescriptor = new SecurityTokenDescriptor
                {
                    Subject = new ClaimsIdentity(new[]
```

```
                    {
                        new Claim(ClaimTypes.Name, login.Username)
                    }),
                    Expires = DateTime.UtcNow.AddHours(1),
                    Issuer = _configuration["Jwt:Issuer"],
                    Audience = _configuration["Jwt:Audience"],
                    SigningCredentials = new SigningCredentials(new
SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
                };

                var token = tokenHandler.CreateToken(tokenDescriptor);
                var jwtToken = tokenHandler.WriteToken(token);

                return Ok(new { Token = jwtToken });
            }

            return Unauthorized();
        }
    }

    public class LoginModel
    {
        public string Username { get; set; }
        public string Password { get; set; }
    }
}
```

---

11. Probar la API con Swagger y Postman

1. Ejecuta el proyecto.
2. En Swagger (`https://localhost:5001/swagger`) primero llama al endpoint `/api/auth/login` con:

```
{
  "username": "admin",
  "password": "password"
}
```

3. Copia el token JWT recibido.

4. En Swagger, pulsa el botón "Authorize" (arriba) y pega el token con prefijo `Bearer `.
5. Ahora puedes llamar a los endpoints protegidos `/api/productos` con autorización.

En Postman puedes hacer lo mismo usando la cabecera:

```
Authorization: Bearer {tu_token_jwt}
```

---

12. Resumen y siguientes pasos

- Hemos creado una API RESTful básica con CRUD para productos.
- Documentamos la API con Swagger.
- Añadimos autenticación JWT para proteger la API.
- Implementamos un endpoint simple para obtener el token.
- Puedes mejorar esta API con:
  - Gestión de usuarios real (base de datos).
  - Validación de datos más robusta.
  - Más claims en el JWT.
  - Tests unitarios e integración.
  - Despliegue en nube o servidor.

Lección 34: Videojuego sencillo en consola - Aventura de texto interactiva en C#

Objetivo

Crear un videojuego simple de aventura en modo texto usando la consola de C#. El jugador podrá elegir opciones para avanzar en la historia y tomar decisiones que cambian el rumbo del juego.

Contenidos

1. **Conceptos clave**
2. **Estructura del programa**
3. **Elementos del juego**
4. **Implementación paso a paso**
5. **Mejoras y extensiones**

---

1. Conceptos clave

- **Consola**: Programa que interactúa con el usuario a través de texto.
- **Entrada y salida**: Usaremos `Console.ReadLine()` para leer la opción del jugador y `Console.WriteLine()` para mostrar texto.
- **Control de flujo**: Condicionales (`if`, `switch`) y bucles para manejar la historia y decisiones.
- **Variables y tipos**: Para almacenar el estado del jugador o decisiones.
- **Funciones / Métodos**: Para organizar el código en partes reutilizables.
- **Estructuras de datos simples**: Para representar escenarios y opciones.

---

2. Estructura del programa

Una aventura de texto básica consta de:

- **Introducción**: Presentar la historia.
- **Escenarios o habitaciones**: Lugares o eventos donde el jugador decide qué hacer.
- **Opciones**: Alternativas que el jugador puede elegir.
- **Estado**: Guardar información relevante (ej. vida, objetos).

- **Finales múltiples**: Según las decisiones, el juego puede terminar distinto.

---

3. Elementos del juego

Vamos a crear:

- Un personaje jugador con una vida inicial.
- Un par de escenarios con opciones.
- Decisiones que afectan la vida o el progreso.
- Posibilidad de ganar o perder según las elecciones.

---

4. Implementación paso a paso

Paso 1: Crear el proyecto

Abre Visual Studio o cualquier editor compatible y crea un proyecto de consola en C#.

Paso 2: Escribir el esqueleto principal

```
using System;

class AventuraTexto
{
    static void Main()
    {
        Console.Title = "Aventura de texto - Misión en la cueva";
        Console.WriteLine("¡Bienvenido a la aventura!");
        Console.WriteLine("Presiona Enter para comenzar...");
        Console.ReadLine();

        // Llamada a la función principal del juego
        Juego();
    }

    static void Juego()
    {
        // Aquí irá la lógica principal
    }
}
```

Paso 3: Crear variables de estado

En el método `Juego`, definimos variables para el estado del jugador:

```
static void Juego()

{

    int vida = 3;

    bool tieneAntorcha = false;


    Console.WriteLine("Estás frente a la entrada de una cueva oscura...");


    // Continuar la aventura...

}
```

Paso 4: Primer escenario con opciones

Mostramos el escenario y pedimos opción al jugador:

```
Console.WriteLine("¿Quieres entrar a la cueva? (sí/no)");

string respuesta = Console.ReadLine().ToLower();


if (respuesta == "sí" || respuesta == "si")

{

    Console.WriteLine("Entras en la cueva y encuentras una antorcha en el
suelo.");

    Console.WriteLine("¿Quieres recoger la antorcha? (sí/no)");

    respuesta = Console.ReadLine().ToLower();


    if (respuesta == "sí" || respuesta == "si")

    {

        tieneAntorcha = true;

        Console.WriteLine("Has recogido la antorcha. Ahora puedes ver mejor en
la oscuridad.");

    }

    else

    {

        Console.WriteLine("Decides no coger la antorcha.");

    }

}

else
```

```
{
    Console.WriteLine("Decides no entrar y la aventura termina aquí.");
    return;
}
```

Paso 5: Segundo escenario con consecuencias

```
Console.WriteLine("Sigues avanzando y te encuentras con un murciélago
gigante.");


if (tieneAntorcha)
{
    Console.WriteLine("La luz de la antorcha asusta al murciélago y se va
volando.");
}
else
{
    Console.WriteLine("El murciélago te ataca porque no tienes luz para
ahuyentarlo.");
    vida--;
    Console.WriteLine($"Has perdido una vida. Vida restante: {vida}");
    if (vida <= 0)
    {
        Console.WriteLine("Has perdido toda tu vida. Fin del juego.");
        return;
    }
}
```

Paso 6: Final simple

```
Console.WriteLine("Finalmente llegas a una cámara con un cofre del tesoro.");
Console.WriteLine("¿Quieres abrirlo? (sí/no)");
respuesta = Console.ReadLine().ToLower();


if (respuesta == "sí" || respuesta == "si")
{
    Console.WriteLine("¡Felicidades! Has encontrado el tesoro y ganado el
juego.");
}
else
```

```
{
    Console.WriteLine("Decides no abrir el cofre y sales de la cueva con las manos vacías.");
}

Console.WriteLine("Gracias por jugar.");
```

Código completo consolidado

```
using System;

class AventuraTexto
{
    static void Main()
    {
        Console.Title = "Aventura de texto - Misión en la cueva";
        Console.WriteLine("¡Bienvenido a la aventura!");
        Console.WriteLine("Presiona Enter para comenzar...");
        Console.ReadLine();

        Juego();
    }

    static void Juego()
    {
        int vida = 3;
        bool tieneAntorcha = false;
        string respuesta;

        Console.WriteLine("Estás frente a la entrada de una cueva oscura...");
        Console.WriteLine("¿Quieres entrar a la cueva? (sí/no)");
        respuesta = Console.ReadLine().ToLower();

        if (respuesta == "sí" || respuesta == "si")
        {
            Console.WriteLine("Entras en la cueva y encuentras una antorcha en el suelo.");
            Console.WriteLine("¿Quieres recoger la antorcha? (sí/no)");
```

```
            }


            respuesta = Console.ReadLine().ToLower();


            if (respuesta == "sí" || respuesta == "si")
            {
                tieneAntorcha = true;
                Console.WriteLine("Has recogido la antorcha. Ahora puedes ver
mejor en la oscuridad.");
            }
            else
            {
                Console.WriteLine("Decides no coger la antorcha.");
            }


            Console.WriteLine("Sigues avanzando y te encuentras con un
murciélago gigante.");


            if (tieneAntorcha)
            {
                Console.WriteLine("La luz de la antorcha asusta al murciélago y
se va volando.");
            }
            else
            {
                Console.WriteLine("El murciélago te ataca porque no tienes luz
para ahuyentarlo.");
                vida--;
                Console.WriteLine($"Has perdido una vida. Vida restante:
{vida}");
                if (vida <= 0)
                {
                    Console.WriteLine("Has perdido toda tu vida. Fin del
juego.");
                    return;
                }
            }


            Console.WriteLine("Finalmente llegas a una cámara con un cofre del
tesoro.");
            Console.WriteLine("¿Quieres abrirlo? (sí/no)");
```

```
            respuesta = Console.ReadLine().ToLower();

            if (respuesta == "sí" || respuesta == "si")
            {
                Console.WriteLine("¡Felicidades! Has encontrado el tesoro y
ganado el juego.");
            }
            else
            {
                Console.WriteLine("Decides no abrir el cofre y sales de la cueva
con las manos vacías.");
            }
        }
        else
        {
            Console.WriteLine("Decides no entrar y la aventura termina aquí.");
        }

        Console.WriteLine("Gracias por jugar.");
    }
}
```

---

5. Mejoras y extensiones

Si quieres profundizar y mejorar tu juego:

- Añadir más escenarios y decisiones.
- Usar métodos para cada escenario para mejorar la organización.
- Añadir inventario con lista de objetos.
- Añadir combate con enemigos.
- Permitir guardar y cargar la partida.
- Añadir bucles para repetir escenarios o pedir opciones válidas.
- Crear un sistema de puntos o experiencia.

Resumen

- Usamos **`Console.ReadLine()`** y **`Console.WriteLine()`** para interacción.
- Controlamos la historia con condicionales.

- Usamos variables para el estado (vida, objetos).
- Creamos decisiones que afectan el progreso y finales diferentes.
- El programa es un ejemplo básico pero escalable.

Epílogo

Llegar hasta aquí significa mucho más de lo que parece. Significa que has tomado la decisión de aprender, que has dedicado tiempo a enfrentarte a la lógica, a la sintaxis, a los errores de compilación y a los desafíos de construir algo desde cero. Has abierto la puerta al pensamiento algorítmico y al diseño estructurado de soluciones, y eso es algo digno de reconocimiento.

Este libro ha sido solo un primer paso en tu camino como programadora o programador en C#. Has aprendido los fundamentos: variables, estructuras de control, clases, objetos, métodos y una visión inicial de la programación orientada a objetos. Estos conceptos son la base sobre la que podrás construir proyectos más complejos, aplicaciones más útiles, y una carrera cada vez más sólida en el mundo del desarrollo de software.

Sin embargo, el aprendizaje no termina aquí. De hecho, en programación nunca termina. La tecnología cambia, los lenguajes evolucionan, los entornos se transforman, y con ellos, también deben crecer nuestras habilidades. El verdadero poder de lo que has aprendido no está solo en el conocimiento adquirido, sino en la mentalidad de resolución, en la capacidad de buscar, probar, equivocarte y seguir adelante.

Te animo a que sigas practicando. Empieza pequeños proyectos: una agenda, una aplicación de notas, un juego simple, un conversor de unidades. Investiga sobre bases de datos, interfaces gráficas, desarrollo web o aplicaciones móviles. Descubre las bibliotecas que C# ofrece, experimenta con .NET, y sobre todo, diviértete creando.

La programación no es solo una habilidad técnica: es una forma de pensar, una herramienta para transformar ideas en realidad. Ahora tienes las llaves. El siguiente paso depende de ti.

Nos vemos en el próximo nivel.